ABSTRACT

Title of dissertation: PLANETARY SCALE DATA STORAGE

Benjamin Bengfort
Doctor of Philosophy, 2018

Dissertation directed by: Professor Peter J. Keleher
Department of Computer Science

The success of virtualization and container-based application deployment has fundamentally changed computing infrastructure from dedicated hardware provisioning to on-demand, shared clouds of computational resources. One of the most interesting effects of this shift is the opportunity to localize applications in multiple geographies and support mobile users around the globe. With relatively few steps, an application and its data systems can be deployed and scaled across continents and oceans, leveraging the existing data centers of much larger cloud providers.

The novelty and ease of a global computing context means that we are closer to the advent of an Oceanstore, an Internet-like revolution in personalized, persistent data that securely travels with its users. At a global scale, however, data systems suffer from physical limitations that significantly impact its consistency and performance. Even with modern telecommunications technology, the latency in communication from Brazil to Japan results in noticeable synchronization delays that violate user expectations. Moreover, the required scale of such systems means that failure is routine.

To address these issues, we explore consistency in the implementation of distributed logs, key/value databases and file systems that are replicated across wide areas. At the core of our system is hierarchical consensus, a geographically-distributed consensus algorithm that provides strong consistency, fault tolerance, durability, and adaptability to varying user access patterns. Using hierarchical consensus as a backbone, we further extend our system from data centers to edge regions using federated consistency, an adaptive consistency model that gives satellite replicas high availability at a stronger global consistency than existing weak consistency models.

In a deployment of 135 replicas in 15 geographic regions across 5 continents, we show that our implementation provides high throughput, strong consistency, and resiliency in the face of failure. From our experimental validation, we conclude that planetary-scale data storage systems can be implemented algorithmically without sacrificing consistency or performance.

PLANETARY SCALE DATA STORAGE

by

Benjamin Bengfort

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Peter J. Keleher, Chair/Advisor
Professor Dave Levin
Professor Amol Deshpande
Professor Daniel Abadi
Professor Derek C. Richardson

Preface

If needed.

Foreword

If needed.

Dedication

To Irena and Henry.

Acknowledgments

I could not have done this alone.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

EC Eventual consistency
HC Hierarchical consensus

Chapter 1: Introduction

Eighteen years ago the Oceanstore paper [1] presented a vision for a data utility infrastructure that spanned the globe. The economic model was that of a cooperative utility provided by a confederation of companies that could buy and sell capacity to directly support their users, with regional providers like airports and cafes installing servers to enhance performance for a small dividend of the utility. This economic model meant that Oceanstore's requirements centered around an untrusted infrastructure to support nomadic data: connectivity, security, durability, and location agnostic storage. To meet these requirements, the Oceanstore architecture was composed of two tiers: pools of byzantine quorums that made localized consistency and placement decisions, along with an optimistic dissemination tree layer that moved data between quorums as correctly as possible without providing guarantees. This architecture, along with a reliance on encryption and key-based access control, could facilitate grid computing storage, a truly decentralized and independent participation of heterogenous computational resources across the globe [2].

Unforeseen by Oceanstore, however, was a fundamental shift in how companies and users accessed computing infrastructure. Improvements in virtualization

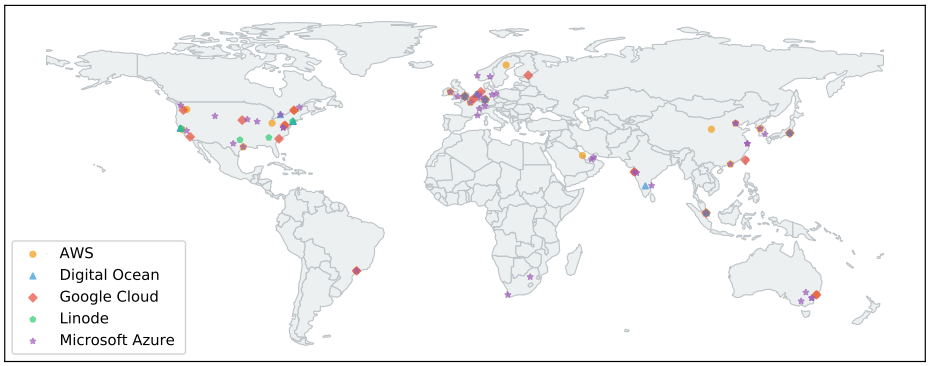

Figure 1.1: Data center locations of popular cloud providers span the globe.

management [3] and later container computing [4] allowed big internet companies to easily lease their unused computational resources and disk capacity to application developers, making cloud computing [5] rather than independent hardware purchasing and hosting the norm. Furthermore, from smarter phones to tablets and netbooks that do not have large disk capacities, user devices have become increasingly mobile; from photos to email and contacts, most user data is now stored in cloud silos. The cloud economy means that there exists a *trusted infrastructure* of virtual resources that span globe, provisioned by a single provider as shown in Figure 1.1.

So what does this mean for the requirements and assumptions of Oceanstore? First, the strict requirements for security that meant per-user encryption and byzantine agreement between untrusted servers can be relaxed to application and transport-level encryption and non-byzantine consensus supported by authenticated communication. Second, the requirements for performance have dramatically increased as ubiquitous computing has become the norm and as more non-human users are par-

ticipating in networks. Increased capacity, however, cannot come at the cost of correctness or consistency, and the increased rate of requests means that asynchronous commits and conflict resolution become far more difficult. For these reasons, we believe that a vision for an Oceanstore today would focus on *consistency* rather than security.

Consistent behavior in distributed systems becomes increasingly complex to implement and reason about as the system size grows and requires increased coordination. In 2021, Cisco forecasts over 25 billion devices will contribute to 105,800 GBps of global internet traffic, 26% of which will be file sharing and application data, and 51% of which will originate from machine to machine-to-machine applications [6]. New types of networks including sensor networks, smart grid solutions, self-driving vehicle networks, and an internet of things will mean an update model with many publishers, few subscribers, and increasingly distributed accesses. To support this growth and facilitate speed, traffic is consistently moving closer to the edge; Cisco predicts that cross-country delivery will drop from 58% of traffic in 2016 to 41% in 2021 and that metro delivery will grow from 22% to 35%. Localization means that the cloud will be surrounded by a fog of devices that participate in systems by contributing data storage and computation to an extent greater than access-oriented clients might.

Based on these trends and inspired by the work of Oceanstore, we propose that a consistent planetary-scale data storage system would be made up of a two tier architecture of both cloud and fog infrastructure. The first tier, in the cloud, would be a strong consistency, fault tolerant and highly resilient geo-replicated con-

sensus backbone: *hierarchical consensus*. The second tier, via the fog, would be a high availability heterogenous network with a hybrid consistency model: *federated consistency*. Such a system would be difficult to manually manage, therefore the system would also have to automatically monitor and adapt to changes in access patterns and node and network availability during runtime. We believe that the combination of hierarchical consensus, federated consistency, and adaptive monitoring lay out a foundation for truly large scale data storage systems that span the planet.

The contributions of this dissertation are therefore as follows:

1. We present the design, implementation, and evaluation of hierarchical consensus, a consensus protocol that can scale to dozens or hundreds of replicas across the wide area.

2. We also investigate the design and implementation of federated consistency, a hybrid consistency model that allows strong, consensus-based systems to integrate with eventually-consistent, highly available replicas and evaluate it in a simulated heterogenous network.

3. We show the possibilities for machine learning-based system adaptation with a reinforcement learning approach to anti-entropy synchronizations based on accesses.

4. We validate our system by describing the implementation of a planetary-scale key-value data store and file system using both hierarchical consensus and federated consistency.

The rest of this dissertation is organized as follows. In the next chapter we will more thoroughly describe the motivations and challenges of building geo-replicated data systems as well as explore case-studies of existing systems. Next, we will focus on the core backbone of our system: hierarchical consensus and describe a globally fault tolerant approach to managing accesses to objects in the wide area. Using hierarchical consensus as a building block, we will next describe federated consistency and how a hybrid, heterogenous consistency model in the fog interacts with the cloud consensus tier. At this point we will have enough background to introduce our system implementation and describe our file system and key-value store. From there, we will explore learning systems that monitor and adapt the performance of the system at runtime, before concluding with related work and a discussion of our future research.

Chapter 2: Challenges and Motivations

Many of the world's most influential companies grew from the ashes of the dot-com bubble of the 1990s, which paid for an infrastructure of fiber-optic cables, giant server farms, and research into mobile wireless networks [7]. As these companies filled market voids in eCommerce, search, and social networking, they created new database technologies to leverage the potential of underused computational resources and low latency/high bandwidth networks that connected them, eschewing more mature systems that were developed with resource scarcity in mind [8, 9]. What followed was the rise and fall of NoSQL data systems, a microcosm of the proceeding era of database research and development [10]. Although there are a lot of facets to the story of NoSQL, what concerns us most is the use of NoSQL to create geographically distributed systems, as these systems paved the way to the large-scale storage systems in use today.

The first phase of distributed NoSQL systems was the creation of highly available, sharded systems intended to meet the demand of increasing numbers of clients. Commercially, these types of systems include Dynamo [11] and BigTable [12], which in turn spawned open source and academic derivatives such as Cassandra [13] and HBase [14]. Although these systems did support large number of accesses, they

achieved their availability by relaxing consistency, which many applications found to be intolerable. The second phase was, therefore, a return to stronger consistency, even at the cost of decreased performance or expensive engineering solutions. Again, commercial systems led the way with Megastore [15] and Spanner [16] along with academic solutions such as MDCC [17] and Calvin [18, 19]. Part of this realignment was a reconsidering of the base assumption that drove the NoSQL movement expressed in the CAP theorem [20], primarily that the lines between availability, partition tolerance, and consistency may not be as strictly drawn as previously theorized [21, 22]. This has led to the beginning of a third phase, the return of SQL, as the lessons learned during the glut of computational resources are applied to more traditional systems. As before, both commercial systems, such as Aurora [23] and Azure SQL [24], and open source systems such as Vitess [25] and CockroachDB [26] are playing an important role in framing the conversation about consistency in this phase.

This brief and limited description of the history of NoSQL distributed systems serves to set the tone for two primary points. First, designing distributed systems to support a large number of users across large geographies entails a number of challenges and trade-offs due to physical limitations that no single architecture has been able to fully cover. Second, there exists commercial and practical motivations to build such systems and these motivations have been the impulse toward academic research. With this backdrop in mind, we explore in this chapter the question of whether a planetary-scale data system is really necessary, and the challenges that we face when designing such systems.

2.1 A New Application Development Paradigm

The launch of the augmented reality game Pokémon GO in the United States was an unmitigated disaster [27]. Due to extremely overloaded servers from the release's extreme popularity, users could not download the game, login, create avatars, or find augmented reality artifacts in their locales. The company behind the platform, Niantic, scrambled quickly, diverting engineering resources away from their feature roadmap toward improving infrastructure reliability. The game world was hosted by a suite of Google Cloud services, primarily backed by the Cloud Datastore [28], a geographically distributed NoSQL database. Scaling the application to millions of users therefore involved provisioning extra capacity to the database by increasing the number of shards as well as improving load balancing and autoscaling of application logic run in Kubernetes [29] containers.

Niantic's quick recovery is often hailed as a success story for cloud services and has provided a model for elastic, on demand expansion of computational resources. A deeper examination, however, shows that Google's global high speed network was at the heart of ensuring that service stayed stable as it expanded [30]. The original launch of the game was in 5 countries – Australia, New Zealand, the United States, the United Kingdom, and Germany; however the success of the game meant worldwide demand, and it was subsequently expanded to over 200 countries starting with Japan [31]. Unlike previous games that were restricted with region locks [32], Pokémon GO was a truly international phenomenon and Niantic was determined to allow international interactions in the game's feature set, interaction which relies on

Google's unified international architecture and globally distributed databases.

In the brief history of NoSQL that started this chapter, we showed that the growth of database systems distributed across the wide-area started with large internet companies like Yahoo, Google, and Amazon but quickly led to academic investigations. One reason that the commercial systems enjoyed this academic attention was that at the time, the unique scale of their usage proved the motivation behind their architecture, however their success has meant that these types of scales are no longer limited to huge software systems. Instead, stories such Niantic's deployment are becoming common and medium to large applications now require developers to increasingly reason about how data is distributed in the wide area, different political regions, and replicated for use around the world.

Consider the following companies and applications from large to small that have international audiences. Dropbox has users in over 180 countries and is supported in 20 languages, maintaining offices in 12 locations from Herzliya to Sydney [33]. Slack serves 9 million weekly active users around the world and has 8 offices around the world, prioritizing North America, Europe, and Pacific regions [34]. WeWork provides co-working space in 250+ international applications and uses an app to manage global access and membership [35]. Tile has sold 15 million of its RFID trackers worldwide and locates 3 million unique items a day in 230 countries [36]. Trello, a project management tool, has been translated into 20 languages and has 250 million world-wide users in every country except Tuvalu, their international rollout focused on marketing and localization [37]. Runkeeper [38] and DarkSky [39] are iOS and Android apps that have millions of global users and struggled to make their

services available in other countries, but benefitted from international app stores. Signal and Telegraph, encrypted messaging apps have grown primarily in countries at the top of Transparency International's Corruption Perception Index [40].

None of the applications described above necessarily have geography-based requirements in the same way that an augmented reality or airline reservations application might have, just a large number of users who regularly use the app from a variety of geographic locations. Web developers are increasingly discussing and using container based approaches both for development and small-scale production, web frameworks have built in localization tools that are employed by default, and services are deployed an autoscaling cloud platforms from the start. The new application development paradigm, even for small applications, is to build with the thought that your application will soon be scaling across the globe.

To address this paradigm, cloud service providers have expanded their offerings to include usage of their distributed data stores to application developers. The problem is that distributed systems like Dynamo, BigTable, Megastore, and Spanner will all designed for in-house services with a data model that supported replication and eventual consistency, but makes it difficult for developers to reason about behavior, as we will see in the next section. Not only is there a need for strong consistency semantics, data-location awareness, and geo-replication in distributed data storage systems, there is also the need for a familiar and standardized storage API.

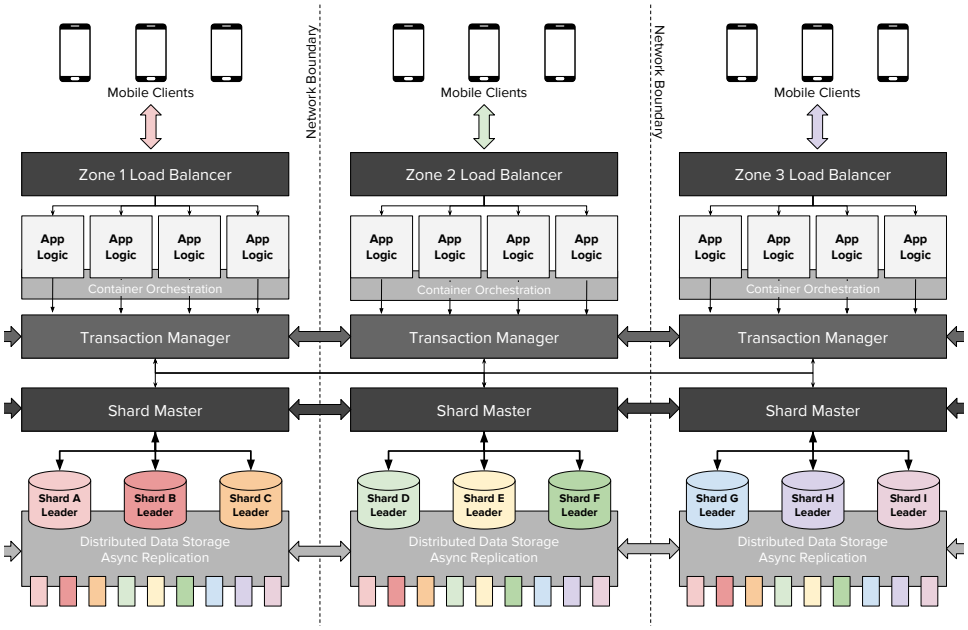

Figure 2.1: A standard architecture of a distributed application.

2.2 Building Geo-Replicated Services

Given non-geographic requirements for building a distributed system, a common architecture that provides high write throughput with consistent replication

During the first phase of

Paxos as the basis for high performance data store [41]

What they didn't do and what we do better.

BigTable & Spanner

S3

Aurora, Cosmos, & Cockroach DB

Approaches: sharding, independent objects, buckets, slow reads

2.3 Requirements for Data Systems

Failure is common Disk failure/replica failure (ODS) Network failure (when repaired, replica comes back online) Unreliability: messages have highly variable latency, out of order messaging Partitions, part of the system cannot speak to the rest of the system

In geo-systems large latency is not the issue! There is a physical limit to message traffic Writes must be applied in order, reads can reason about staleness Access patterns are also location-dependent

Durability Normally 3 disk replication ensures 2 failures We need to ensure zone+1 disk failures (re Aurora) User-specific data should be accessible everywhere

Fault-Tolerance & availability System should still be available if nodes fail Should be available if zones fail (e.g. hurricane or disaster) at higher latency System should be available at lower consistency if even one replica is available

Adaptability Should respond to changes in user access patterns Should be able to add and remove nodes from the system Should scale with more regions and more replicas

2.4 System Architecture

Describe the architecture of tier 1: HC, tier 2: Federated Fog

Describe the consistency guarantees we claim

Base application is a key/value store

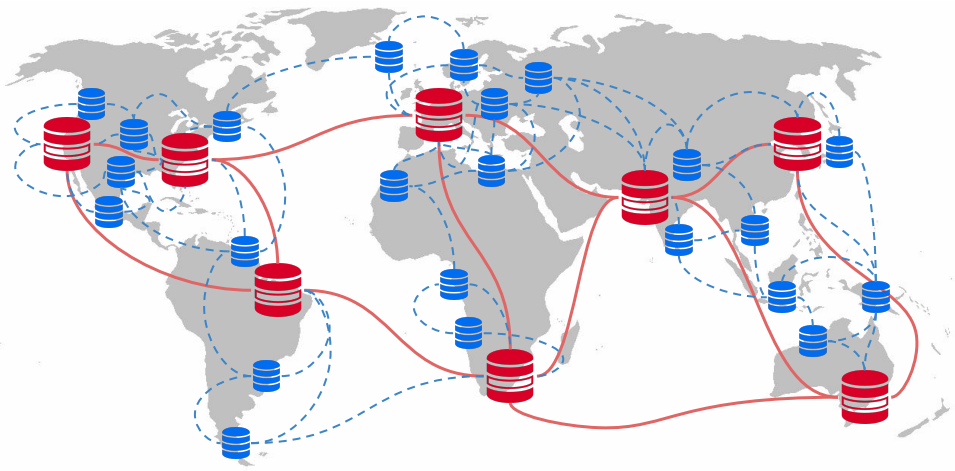

Figure 2.2: A global architecture composed of a core backbone of hierarchical consensus replication (red) and a fog of heterogenous, federated consistency replicas (blue).

File-system is built upon the key/value store as an object FS

In Figure 2.2

2.5 Conclusion

1. Global-scale applications are becoming the new norm

2. Today's globally distributed data systems are high level and require developers to deeply consider consistency and localization semantics.

3. The challenges are around the physical limitations of geographic networks: the speed of light in this case; e.g. latency and outages.

Chapter 3: Hierarchical Consensus

The backbone of our planetary scale data system is *hierarchical consensus* [42]. Hierarchical consensus provides a strong consistency foundation that totally orders critical accesses and arbitrates the eventual consistency layer in the fog, which raises the overall consistency of the system. To be effective, an externalizable view of consistency ordering must be available to the entire system. This means that strong consistency must be provided across geographic links rather than provided as localized, independent decision making with periodic synchronization. The problem that hierarchical consensus is therefore designed to solve is that of distributed consensus.

Solutions to distributed consensus primarily focus on providing high throughput, low latency, fault tolerance, and durability. Current approaches [18,43–47] usually assume a small number of replicas, each centrally located on a highly available, powerful, and reliable host. These assumptions are justified by the environments in which they run: highly curated environments of individual data centers connected by dedicated networks. Although replicas in these environments may participate in global consensus, our data model requires us to accommodate replicas with heterogenous capabilities and usage modalities. Widely distributed replicas might have neither high bandwidth nor low latency and might suffer partitions of varying du-

rations. Such systems of replicas might also be dynamic in membership, in relative locations, and have relative workloads. Most importantly, to provide a backbone for a planetary scale data system, the consistency backbone must scale to include potentially hundreds of replicas around the world.

As a result, straightforward approaches of running variants of Paxos [48], ePaxos [43], or Raft [49] across the wide area, even for individual objects will perform poorly for several reasons. First, distance (in network connectivity) between consensus replicas and the most active replicas decrease the performance of the entire system, consensus is only as fast as the final vote required to make a decision, even when making thrifty requests. Second, network partitions are common, which cause consensus algorithms to fail-stop [50] if they cannot receive a majority, a criticism that is often used to justify eventual consistency systems for high availability. Finally, the fault tolerance of small quorum algorithms can be disrupted by a small number of unreliable hosts and given the scale of the system and the heterogenous nature of replicas, the likelihood of individual failure is so high so as to be considered inevitable.

We propose another approach to building large systems. Rather than relying on a few replicas to provide consensus to many clients, we propose to run a consensus protocol across replicas running at or near all of these locations. The key insight is that large problem spaces can often be partitioned into mostly disjoint sets of activity without violating consistency. We exploit this decomposition property by making our consensus protocol hierarchical and individual consensus groups fast by ensuring they are small. We exploit locality by building subquorums from colocated

replicas, and locating subquorums near clients they serve.

In this chapter we describe hierarchical consensus, a two-tiered consensus structure that allows high throughput, localization, agility, and linearizable access to a shared namespace. We show how to use *delegation* to build large consensus groups that retain their fault tolerance properties while performing like small groups. We describe the use of *fuzzy epoch transitions* to allow global re-configurations across multiple consensus groups without forcing them into lockstep. Finally, we describe how we reason about consistency by describing the structure of grid consistency.

3.1 Overview

Hierarchical Consensus (HC) is an implementation and extension of Vertical Paxos [51–53] designed to scale to hundreds of nodes geo-replicated around the world. Vertical Paxos divides consensus decisions both horizontally, as sequences of consensus instances, and vertically as individual consensus decisions are made. Spanner [16], MDCC [17], and Calvin [18], can all be thought of as implementations of Vertical Paxos, in that they shard the namespace of the objects they manage into individual consensus groups (the vertical division). In these cases, however, sharding does not allow for inter-object dependence (the horizontal division) without either a management quorum which is either not geo-replicated, suffers from the same problems in scaling, or without the use of extremely accurate timestamps. The challenge is therefore in building a multi-group coordination protocol that configures and mediates the subquorums with the same level of consistency and fault tolerance

of the entire system.

Hierarchical consensus therefore organizes *all* participating replicas to participate in a root quorum as shown in Figure 3.1. The root quorum guarantees correctness by pivoting the overall system through two primary functions. First, the root quorum reconfigures subquorum memberships on replica failures and system membership changes (allocating hot-spares as needed). Second, the root quorum adjusts the mapping of the object namespace to the underlying partitions. Much of the system's complexity comes from handshaking between the root quorum and the lower-level subquorums during reconfigurations.

These handshakes are made easier, and much more efficient, by using *fuzzy transitions*. Fuzzy transitions allow individual subquorum to move through reconfiguration at their own pace, allowing portions of the system to transition to decisions made by the root quorum before others. Given out heterogenous, wide-area environment, forcing the entire system to transition to new configurations in lockstep would be unacceptably slow. Fuzzy transitions also ensure that there is no dedicated shard-master that has to synchronize all namespace allocations: at the cost of possibly multiple redirections, clients can be redirected by any member of the root quorum to replicas who should be participating in consensus decisions for the requested objects.

Fuzzy transitions ensure that root quorum decisions need not be timely since those decisions do not disrupt accesses of clients. Though root quorum decisions are rare with respect to the throughput of accesses inside the entire system, they still do require the participation of all members of the system, which could lead

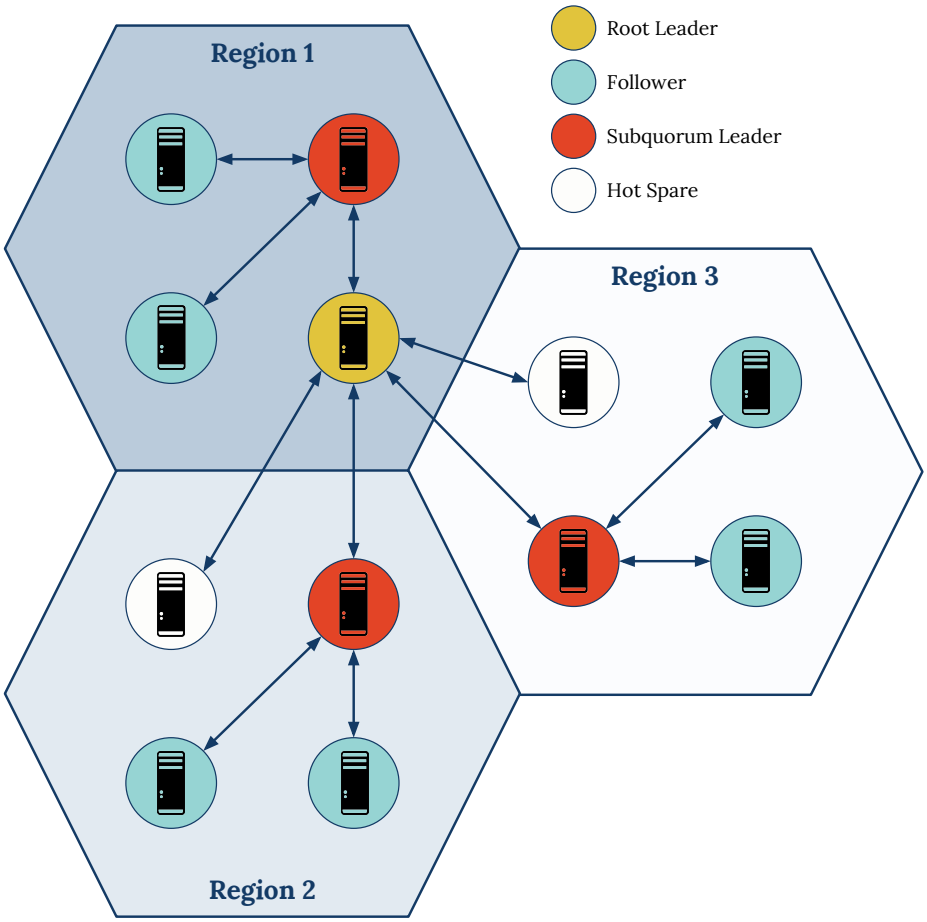

Figure 3.1: A simple example of an HC network composed of 12 replicas with size 3 subquorums. Each region hosts its own subquorum and subquorum leader, while the subquorum leaders delegate their votes to the root quorum, whose leader is found in region 1. This system also has 2 hot spares that can be used to quickly reconfigure subquorums that experience failures. The hot spares can either delegate their vote, or participate directly in the root quorum.

to extremely large quorum sizes, and therefore extremely slow consensus operations that may be extremely sensitive to partitions. Because all subquorums make disjoint decisions and because all members of the system are part of the root quorum, we propose a safe relaxation of the participation requirements for the root quorum such that subquorum followers can *delegate* their root quorum votes to their leader. Delegation ensures that only a small number of replicas participate in most root quorum decisions, though decisions are made for the entire system.

In brief, the resulting system is local, in that replicas serving clients can be located near them. The system is fast because individual operations are served by small groups of replicas, regardless of the total size of the system. The system is nimble in that it can dynamically reconfigure the number, membership, and responsibilities of subquorums in response to failures, phase changes in the driving applications, or mobility among the member replicas. Finally, the system is consistent, supporting the strongest form of per-object consistency without relying on special-purpose hardware [16, 54–57].

A complete summary of hierarchical consensus is described in Figure 3.2.

3.2 Consensus

The canonical distributed consensus used by systems today is Paxos [48, 58]. Paxos is provably safe and designed to make progress even when a portion of the system fails. Raft [49] was designed not to improve performance, but to increase understanding of consensus behavior to better allow efficient implementations. HC

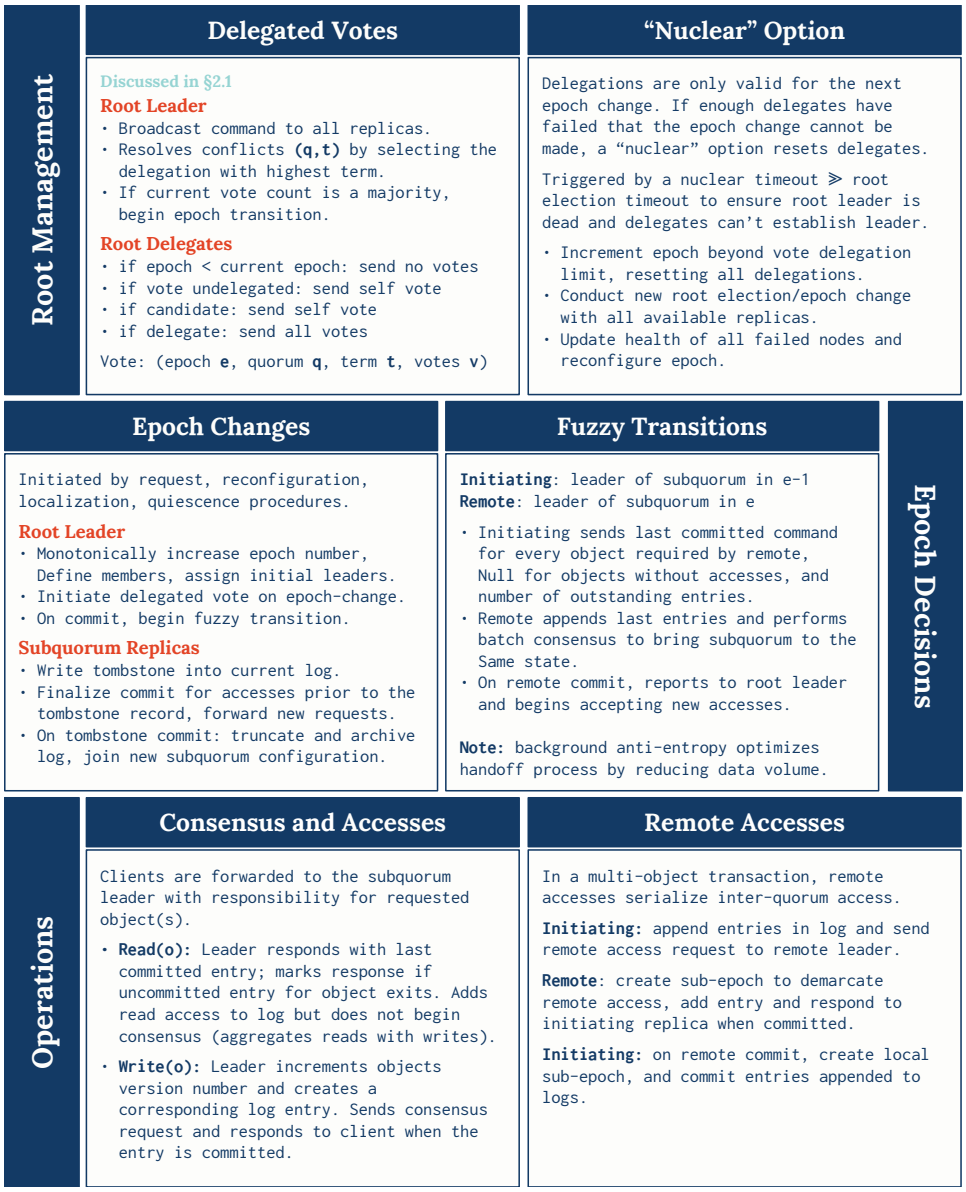

Figure 3.2: A condensed summary of the hierarchical consensus protocol. Operations are described in a top-to-bottom fashion where the top level is root quorum operations, the bottom is subquorum operations, and the middle is transition and intersection.

uses Raft as a building block, so we describe the relevant portions of Raft at a high level, referring the reader to the original paper for complete details. Though we chose to base our protocol on Raft, a similar approach could be used to modify Paxos into a hierarchical structure.

Consensus protocols typically have two phases: leader *election* and operations *commit* [1] . Raft is a strong-leader consensus protocol, which allows the election phase to be elided while a leader remains available. The protocol requires only a single communication round to commit an operation in the common case. Raft uses timeouts to trigger phase changes and provide fault tolerance. Crucially, it relies on timeouts only to provide progress, not safety. New elections occur when another replica in the quorum times out waiting for communication from the leader. Such a replica increments its *term* until it is greater than the existing leader, and announce its candidacy. Other replicas vote for the candidate if they have not seen a competing candidate with a larger term. During regular operation, clients send requests to the leader, which broadcasts `AppendEntries` messages carrying operations to all replicas. An operation is *committed* and can be executed when the leader receives acknowledgments of the `AppendEntries` message from more than half the replicas (including itself).

We describe differences in our Raft implementation from the canonical implementation in Chapter 5

Throughout the rest of this chapter we use the term *root quorum* to refer to the upper, namespace-mapping and configuration-management tier of HC, and

[1] Election and commit phases correspond to PROPOSE and ACCEPT phases in Paxos

subquorum to describe a group of replicas (called *peers*) participating in consensus decisions for a section of the namespace. The root quorum shepherds subquorums through *epochs*, each with potentially different mappings of the namespace and replicas to subquorums. An epoch corresponds to a single commit phase of the root quorum. We use the term Raft only when describing details particular to our current use of Raft as the underlying consensus algorithm. We refer to the two phases of the base consensus protocol as the *election phase* and the *commit phase*. We use the term *vote* as a general term to describe positive responses in either phase. Epoch x is denoted e_x. Subquorum i of epoch e_x is represented as $q_{i,x}$, or just q_i when the epoch is obvious. t_a represents a specific *tag*, or disjoint subset of the namespace.

We assume faults are fail-stop [50] rather than Byzantine [59]. We do not assume that either replica hosts or networks are homogeneous, nor do we assume freedom from partitions and other network faults.

3.2.1 Root Consensus

Hierarchical consensus is a leader-oriented protocol that organizes replicas into two tiers of quorums, each responsible for fundamentally different decisions (Figure 3.3). The lower tier consists of multiple independent subquorums, each committing operations to local shared logs. The upper, *root quorum*, consists of subquorum peers, usually their leaders, delegated to represent the subquorum in root elections and commits. Hierarchical consensus's main function is to export a linearizable abstraction of shared accesses to some underlying substrate, such as a

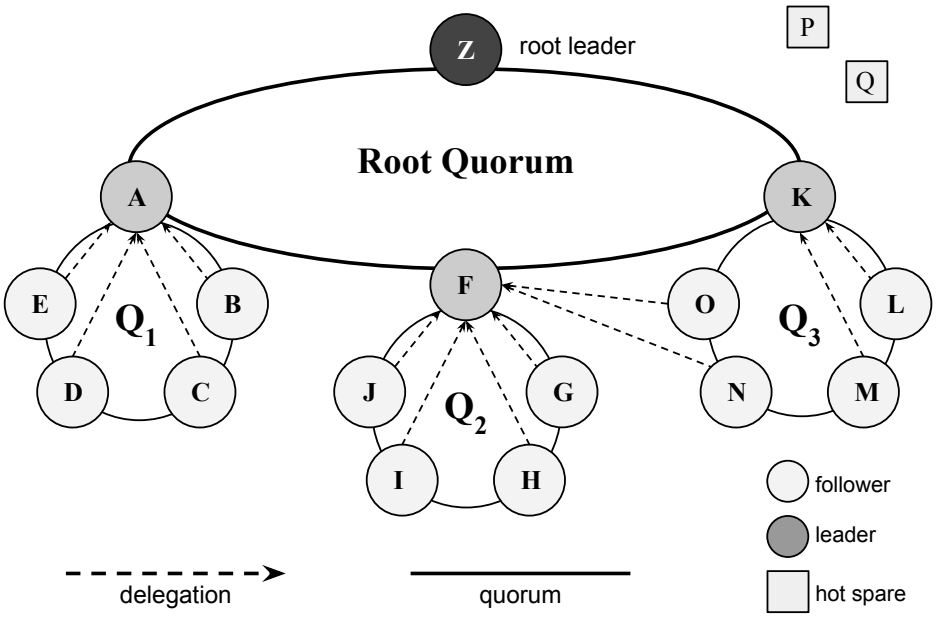

Figure 3.3:

distributed object store or file system. We assume that nodes hosting object stores, applications, and HC are frequently co-located across the wide area.

The root quorum's primary responsibilities are mapping namespaces and replicas to individual subquorums. Each such map defines a distinct epoch, e_i, a monotonically increasing representation of the term of $q_{i,e}$. The root quorum is effectively a consensus group consisting of subquorum leaders. Somewhat like subquorums, the effective membership of the root quorum is not defined by the quorum itself, but in this case by leader election or peer delegations in the lower tier.

The root quorum partitions the namespace across multiple subquorums, each with a disjoint portion as its scope. The intent of subquorum localization is ensure that the *domain* of a client, the portion of the namespace it accesses, is entirely

within the scope of a local, or nearby, subquorum. If true across the entire system, each client interacts with only one subquorum, and subquorums do not interact at all during execution of a single epoch. This *siloing* of client accesses simplifies implementation of strong consistency guarantees and allows better performance.

3.2.2 Delegation

Fault tolerance scales with increasing system size. The root quorum's membership is, at least logically, the set of all system replicas, at all times. However, running consensus elections across large systems is inefficient in the best of cases, and prohibitively slow in a geo-replicated environment. Root quorum decision-making is kept tractable by having replicas *delegate* their votes, usually to their leaders, for a finite duration. With leader delegation, the root membership effectively consists of the set of subquorum leaders. Each leader votes with a count describing its own and peer votes from its subquorum.

Consider an alternative leader-based approach where root quorum membership is defined as the current set of subquorum leaders. Both delegation and the leader approach have clear advantages in performance and flexibility over direct votes of the entire system. However, the leader approach dramatically decreases fault tolerance. Furthermore, the root quorum becomes unstable in the leader approach as its membership changes during partitions or subquorum elections. These changes would require heavyweight *joint consensus* decisions in the root quorum for correctness in Raft-like protocols [49].

With delegation, however, root quorum membership is always the entire system and remains unchanged over subquorum re-configuration. Delegation is essentially a way to optimistically shortcut contacting every replica for each decision. Subquorum repartitioning merely implies that a given replica's vote might need to be delegated to a different leader.

Delegation does add one complication: the root quorum leader must know all vote delegations to request votes when committing epoch changes. We deal with this issue, as well as the requirement for a nuclear option (Section 3.4.3), by simplifying our protocol. Instead of sending vote requests just to subquorum leaders, **the root quorum leader sends vote requests to all system replicas.** This is true even for *hot spares*, which are not currently in any subquorum.

This is correct because vote requests now reach all replicas, and because replicas whose votes have been delegated merely ignore the request. We argue that it is also efficient, as a commit's efficiency depends only on receipt of a majority of the votes. Large consensus groups are generally slow (see Section 3.5) not just because of communication latency, but because large groups in a heterogeneous setting are more likely to include replicas on very slow hosts or networks. In the usual case for our protocol, the root leader still only needs to wait for votes from the subquorum leaders. Leaders are generally those that respond more quickly to timeouts, so the speed of root quorum operations is unchanged.

3.2.3 Epoch Transitions

An epoch change is initiated by the leader in response to one of several events, including:

- a namespace repartition request from a subquorum leader
- notification of join requests by new replicas
- notification of failed replicas
- changing network conditions that suggest re-assignment of replicas

The root leader transitions to a new epoch through the normal commit phase in the root quorum. The command proposed by the leader is an enumeration of the new subquorum partition, namespace partition, and assignment of namespace portions to specific subquorums. The announcement may also include initial leaders for each subquorum, with the usual rules for leader election applying otherwise, or if the assigned leader is unresponsive. Upon commit, the operation serves as an *announcement* to subquorum leaders. Subquorum leaders repeat the announcement locally, disseminating full knowledge of the new system configuration, and eventually transition to the new epoch by committing an `epoch-change` operation locally.

The epoch change is lightweight for subquorums that are not directly affected by the underlying re-configuration. If a subquorum is being changed or dissolved, however, the *epoch-change* commitment becomes a tombstone written to the logs of all local replicas. No further operations will be committed by that version of the

subgroup, and the local shared log is archived and then truncated. Truncation is necessary to guarantee a consistent view of the log within a subquorum, as peers may have been part of different subquorums, and thus have different logs, during the last epoch. Replicas then begin participating in their new subquorum instantiation. In the common case where a subquorum's membership remains unchanged across the transition, an `epoch-change` may still require additional mechanism because of changes in namespace responsibility.

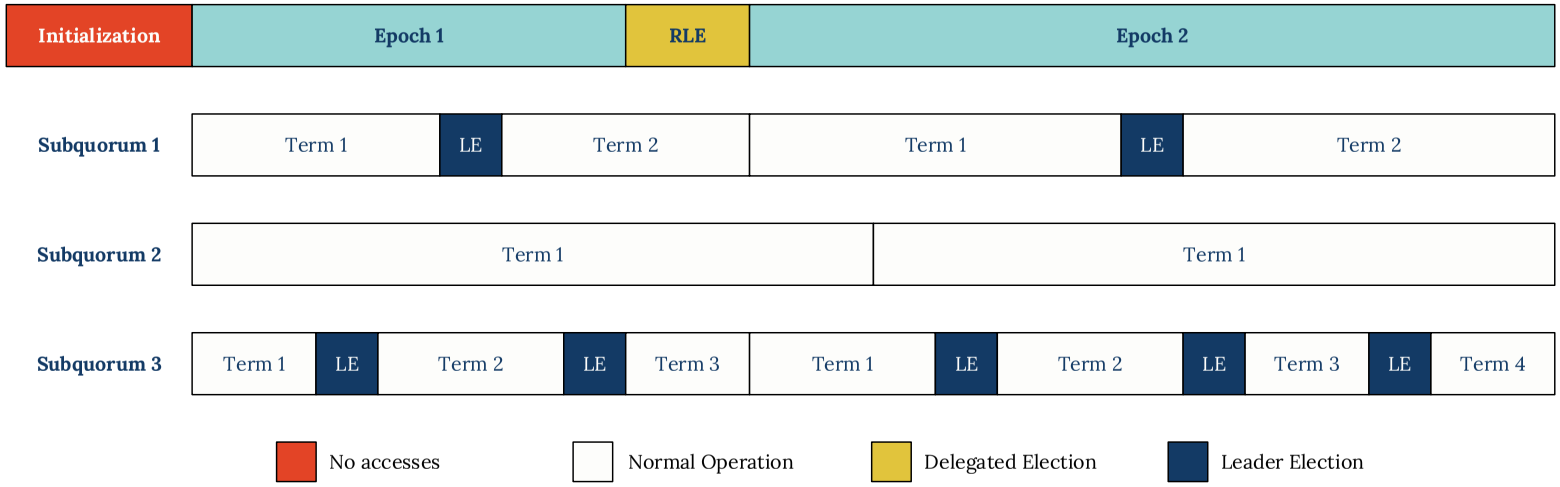
Initialization
Epoch 1
RLE
Epoch 2
Subquorum 1
Term 1
LE
Term 2
Term 1
LE
Term 2
Subquorum 2
Term 1
Term 1
Subquorum 3
Term 1
LE
Term 2
LE
Term 3
Term 1
LE
Term 2
LE
Term 3
LE
Term 4
No accesses
Normal Operation
Delegated Election
Leader Election

Figure 3.4:

3.2.4 Fuzzy Handshakes

Epoch handshakes are required whenever the namespace-to-subquorum mapping changes across an epoch boundary. HC separates epoch transition announcements in the root quorum from implementation in subquorums. Epoch transitions are termed *fuzzy* because subquorums need not all transition synchronously. There are many reasons why a subquorum might be slow. Communication delays and partitions might delay notification. Temporary failures might block local commits. A subquorum might also delay transitioning to allow a local burst of activity to cease such as currently running transactions [2] . Safety is guaranteed by tracking subquorum dependencies across the epoch boundary.

Figure 3.5 shows an epoch transition where the scopes of q_i, q_j, and q_k change across the transition as follows:

Fix the alignment of the below equations:

$$q_{i,x-1} = t_a, t_b \quad \longrightarrow q_{i,x} = t_a \tag{3.1}$$

$$q_{j,x-1} = t_c, t_d \quad \longrightarrow q_{j,x} = t_c, t_d, t_f \tag{3.2}$$

$$q_{k,x-1} = t_e, t_f \quad \longrightarrow q_{k,x} = t_d, t_e \tag{3.3}$$

All three subquorums learn of the epoch change at the same time, but become ready with varying delays. These delays could be because of network lags or ongoing local activity. Subquorum q_i gains no new tags across the transition and moves

[2]The HC implementation discussed in this chapter does not currently support transactions.

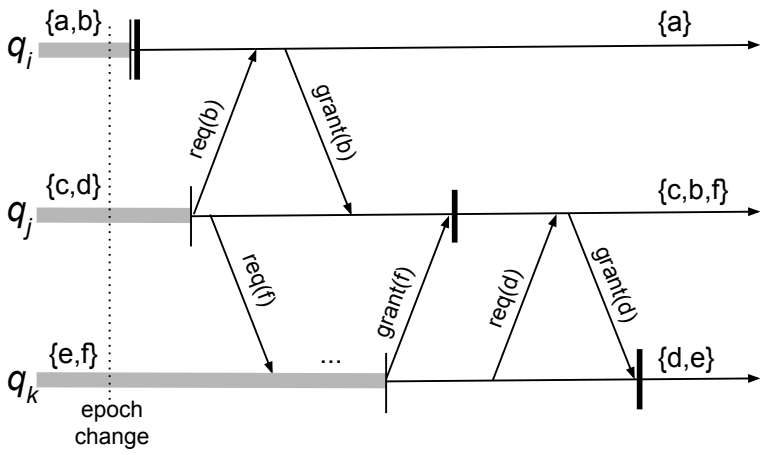

Figure 3.5: Readiness to transition to the new epoch is marked by a thin vertical bar; actual transition is the thick vertical bar. Thick gray lines indicate operation in the previous epoch. Subquorum q_j transitions from tag c, d to c, b, f, but begins only after receiving version information from previous owners of those tags. The request to q_k is only answered once q_k is ready to transition as well.

immediately to the new epoch. Subquorum q_j's readiness is slower, but then it sends requests to the owners of both the new tags it acquires in the new epoch. Though q_i responds immediately, q_k delays its response until locally operations conclude. Once both handshakes are received, q_j moves into the new epoch, and q_k later follows suit.

These bilateral handshakes allow an epoch change to be implemented incrementally, eliminating the need for lockstep synchronization across the entire system. This flexibility is key to coping with partitions and varying connectivity in the wide area. However, this piecewise transition, in combination with subquorum re-definition and configuration at epoch changes, also means that individual replicas *may be part of multiple subquorums at a time.*

This overlap is possible because replicas may be mapped to distinct subgroups from one epoch to the next. Consider q_k in Figure 3.5 again. Assume the epochs shown are e_x and e_{x+1}. A single replica, r_a, may be remapped from subquorum $q_{k,x}$ to subquorum $q_{i,x+1}$ across the transition. Subquorum $q_{k,x}$ is late to transition, but $q_{i,x+1}$ begins the new epoch almost immediately. Requiring r_a to participate in a single subquorum at a time would potentially delay $q_{i,x+1}$'s transition and impose artificial synchronicity constraints on the system. One of the many changes we made in the base Raft protocol is to allow a replica to have multiple distinct shared logs. Smaller changes concern the mapping of requests and responses to the appropriate consensus group.

3.2.5 Subquorum Consensus

this section is bad

Each subquorum, q_i, elects a leader to coordinate local decisions. Fault tolerance of the subquorum is maintained in the usual way, detecting leader failures and electing new leaders from the peers. Subquorums do not, however, ever change system membership on their own. Subquorum membership is always defined in the root quorum.

Subquorum consensus is used to commit object writes. Reads are not committed by default, but are always served by the leader of the appropriate subquorum. Namespace assignments in HC result in the object space being partitioned (or sharded) across distinct subqourms. The mapping of the shared object namespace to individual subquorums is the *tagset*, or the tagset partition. An individual *tag* defines a disjoint subset of the object space.

As writes are committed through HC, the shared logs provide a complete version history of all distributed objects. Subquorum leaders use in-core caches to provide fast access to recently accessed objects in the local subquorums's tag. Replicas perform background anti-entropy [11, 60, 61], disseminating log updates a user-defined number of times across the system.

The most complex portion of the HC protocol is in handling data-related issues at epoch transitions. Transitions may cause tags to be transferred from one subquorum to another, forcing the new leader to load state remotely to serve object requests. Transitions handshakes are augmented in three ways. First, an replica

can demand-fetch an object version from any other system replica. Second, epoch handoffs contain enumerations of all current object versions, though not the data itself. Knowing an object's current version gives the new handler of a tag the ability to demand fetch an object that is not yet present locally. Finally, handshakes start immediate fetches of the in-core version cache from the leader of the tag's subquorum in the old epoch to the leader in the new.

We do not currently gather the entire shared log onto a single replica because of capacity and flexibility issues. Capacity is limited because our system and applications are expected to be long-lived. Flexibility is a problem because HC, and applications built on HC, gain much of their value from the ability to pivot quickly, whether to deal with changes in the environment or for changing application access patterns. We require handoffs to be as lightweight as possible to preserve this advantage.

3.2.6 Client Operations

- Sessions - Connect to closest available replica, redirected to closest available leader.

Client namespace accesses are forwarded to the leader of the subquorum for the appropriate part of the namespace. The underlying Raft semantics ensure that leadership changes do not result in loss of any commits. Hence, individual- or multiple-client accesses to a single subquorum are totally ordered. *Remote accesses*, or client accesses to other than their local subquorum, are transparent to the primary

protocol. However, creation of a single shared log of all system operations requires remote accesses to be logged.

3.3 Consistency

Pushing all writes through subquorum commits and serving reads at leaders allows us to guarantee that accesses are linearizable (Lin), which is the strongest non-transactional consistency [62,63]. As a recap, linearizability is a combination of atomicity and timeliness guarantees about accesses to a single object. Both `reads` and `writes` must appear atomic, and also instantaneous at some time between a request and the corresponding response to a client. `Reads` must always return the latest value. This implies that reads return values are consistent *with any observed ordering*, i.e., the ordering is *externalizable* [64].

Linearizability of object accesses can be *composed*. If operations on each object are linearizable, the entire object space is also linearizable. This allows our subquorums to operate independently while providing a globally consistent abstraction.

3.3.1 Globally Consistent Logs

Our default use case is in providing linearizable access to an object store. Though this approach allows us to guarantee all observers will see linearizable results of object accesses in real-time, the system is not able to enumerate a total order, or create a linearizable shared log. Such a linear order would require fine-grained (expensive) coordination across the entire system, or fine-grained clock synchro-

nization [16]. Though many or most distributed applications (objects stores, file systems, etc.) will work directly with HC, shared logs are a useful building block for distributed systems.

HC *can* be used to build a sequentially consistent (SC) shared log as shown in Figure 3.6. Like Lin, SC requires all observers to see a single total ordering. SC differs in that this total ordering does not have to be externalizable. Instead, it merely has to conform to local operation orders and all reads-from dependencies.

write about grid consistency

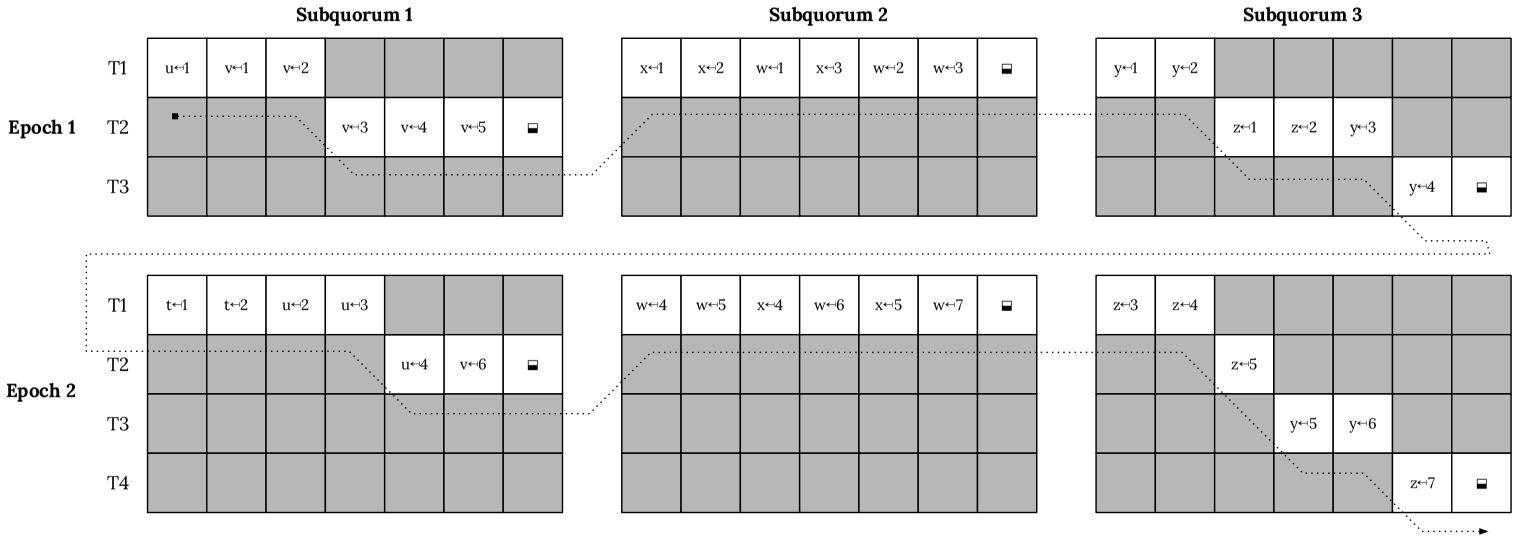
Subquorum 1
Subquorum 2
Subquorum 3
Epoch 1
Epoch 2
T1
T2
T3
T4
u←1
v←1
v←2
v←3
v←4
v←5
x←1
x←2
w←1
x←3
w←2
w←3
y←1
y←2
z←1
z←2
y←3
y←4
t←1
t←2
u←2
u←3
u←4
v←6
w←4
w←5
x←4
w←6
x←5
w←7
z←3
z←4
z←5
y←5
y←6
z←7

Figure 3.6:

Figure 3.7 shows a system with subquorums q_i and q_j, each of which performs a pair of writes. Dotted lines show one possible event ordering for replicas q_i (responsible for objects a and b), and q_j (c and d). Without cross-subquorum reads or writes, ordering either subquorums's operations first creates a SC total ordering: $q_i \rightarrow q_j$ ("happened-before" [65]) implies $w_{i,1} \rightarrow w_{i,3} \rightarrow w_{j,1} \rightarrow w_{j,3}$, for example.

By contrast, the subquorums in Figure 3.8 create additional dependencies by issuing remote writes to other subquorums: $w_{i,2} \rightarrow w_{j,3}$ and $w_{j,2} \rightarrow w_{i,3}$. Each remote write establishes a partial ordering between events of the sender before the sending of the write, and writes by the receiver after the write is received. Similar dependencies result from remote reads.

These dependencies cause the epochs to be split (not shown in picture). The receipt of write $w_{i,2}$ in q_j causes $q_{j,1}$ to be split into $q_{j,1.1}$ and $q_{j,1.2}$. Likewise, the receipt of write $w_{j,2}$ into q_i causes q_i to be split into $q_{i,1.1}$ and $q_{i,1.2}$. Any topological sort of the subepochs that respects these orderings, such as $q_{i,1.1} \rightarrow q_{j,1.1} \rightarrow q_{j,1.2} \rightarrow q_{i,1.2}$, results in a valid SC ordering.

Presenting a sequentially consistent global log across the entire system, then, only requires tracking these inter-subquorum data accesses, and then performing an $\mathcal{O}(n)$ merge of the subepochs.

By definition, this log's ordering respects any externally visible ordering of cross-subquorum accesses (accesses visible to the system). However, the log does not necessarily order other accesses according to external visibility. The resulting shared log could not be mined to find causal relationships between accesses through external communication paths unknown to the system.

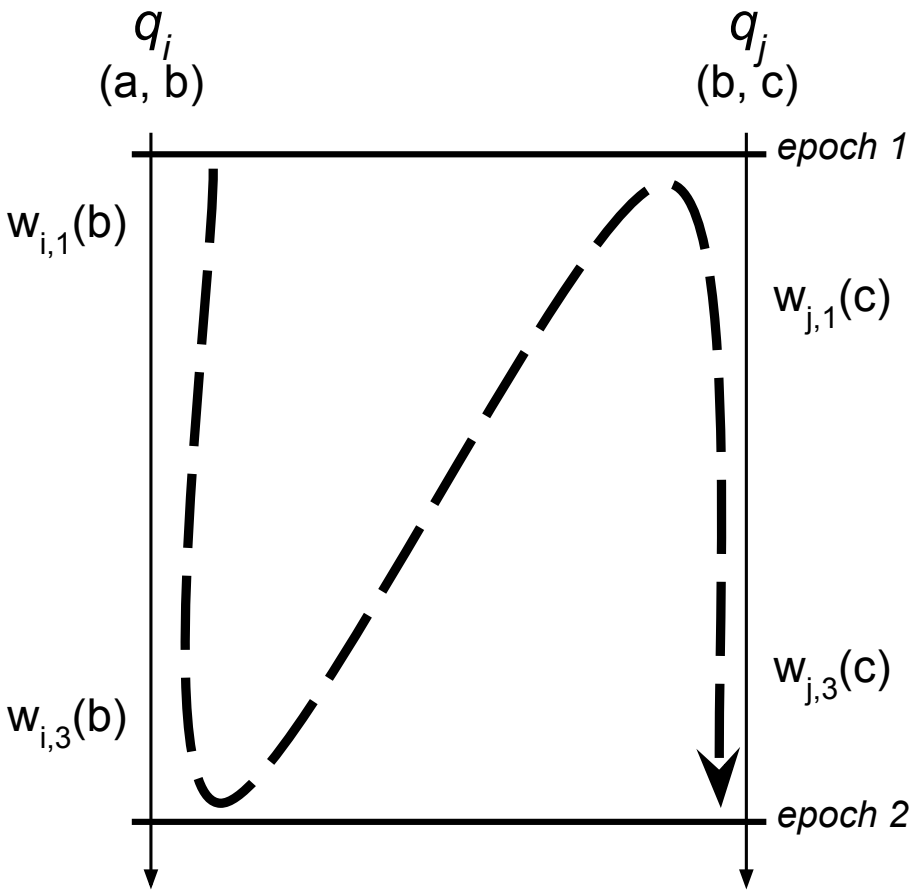

Figure 3.7: Default ordering: $w_{i,1 \to w_{i,3} \to w_{j,1} \to w_{j,3}}$

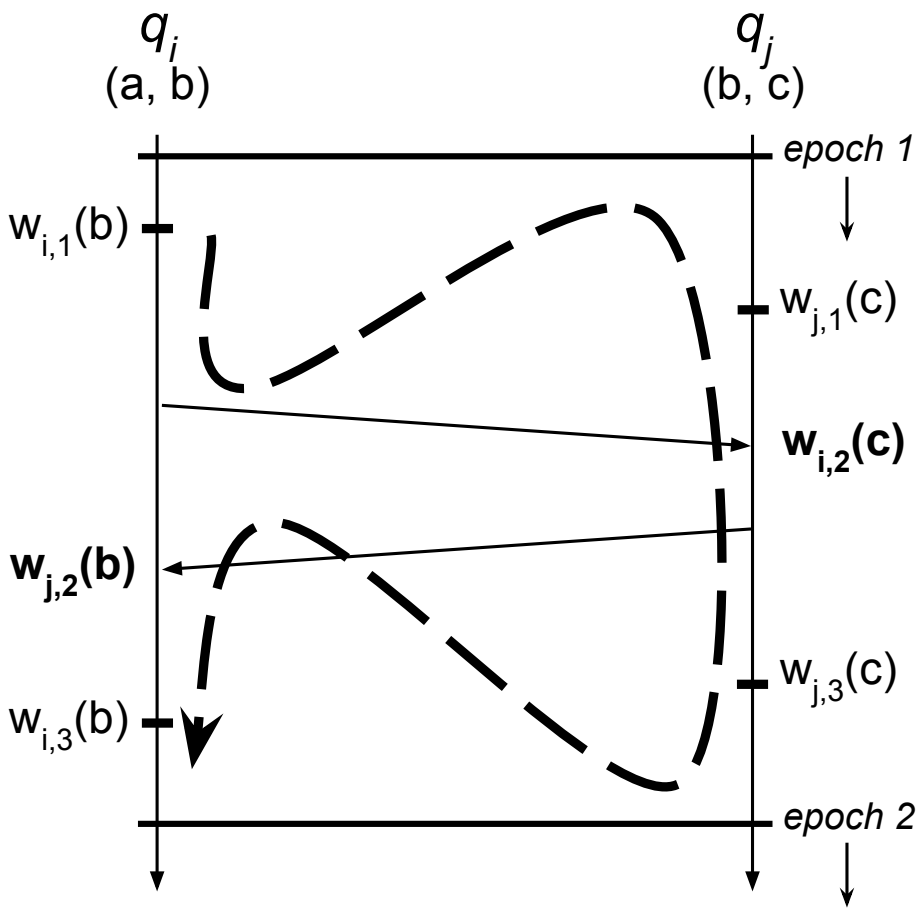

Figure 3.8: Remote writes add additional ordering constraints: $w_{i,1} \rightarrow w_{i,2} \rightarrow w_{j,3}$, and $w_{j,1} \rightarrow w_{j,2} \rightarrow w_{i,3}$

For example, assume that log events are published posts, and that one user claimed plagiarism. The accused would not be able to prove that his post came first unless there were some causal chain of posts and references visible to the protocol.

3.4 Fault Tolerance

We assert that consensus at the leaf replicas is correct and safe because decisions are implemented using well-known leader-oriented consensus approaches. Hierarchical consensus therefore has to demonstrate linearizable correctness and safety between subquorums for a single epoch and between epochs. Briefly, linearizability requires external observers to view operations to objects as instantaneous events. Within an epoch, subquorum leaders serially order local accesses, thereby guaranteeing linearizability for all replicas in that quorum.

Epoch transitions raise the possibility of portions of the namespace being reassigned from one subquorum to another, with each subquorum making the transition independently. Correctness is guaranteed by an invariant requiring subquorums to delay serving newly acquired portions of the namespace until after completing all appropriate handshakes.

Table 3.1: Failure categories: Peer failure is detected by missed heartbeat messages. The rest are triggered by the appropriate election timeout.

Failure Type	Response
subquorum peer	request replica repartition from root quorum
subquorum leader	local election, request replacement from root quorum
root leader	root election (with delegations)
majority of majority of subquorums	(nuclear option) root election after delegations timed out

3.4.1 Failures

During failure-free execution, the root quorum partitions the system into disjoint subquorums, assigns *subquorum leaders*, and assigns partitions of the tagspace to subquorums. Each subquorum coordinates and responds to accesses for objects in its assigned tagspace. We define the system's *safety* property as guaranteeing that non-linearizable (or non-sequentially-consistent, see Section 3.3.1) event orderings can never be observed. We define the system's *progress* property as the system having enough live replicas to commit votes or operations in the root quorum.

The system can suffer several types of failures, as shown in Table 3.1. Failures of subquorum and root quorum leaders are handled through the normal consensus mechanisms. Failures of subquorum peers are handled by the local leader petitioning the root quorum to re-configure the subquorum in the next epoch. Failure of a root quorum peer is the failure of subquorum leader, which is handled as above. Root quorum heartbeats help inform other replicas of leadership changes, potentially necessary when individual subquorums break down.

describe assasination

HC's structure means that some faults are more important than others. Proper operation of the root quorum requires the majority of replicas in the majority of subquorums to be non-faulty. Given a system with $2m+1$ subquorums, each of $2n+1$ replicas, the entire system's progress can be halted with as few as $(m+1)(n+1)$ well-chosen failures. Therefore, in worst case, the system can only tolerate: $f_{worst} = mn + m + n$ failures and still make progress. At maximum, HC's basic protocol

can tolerate up to: $f_{best} = (m+1) * n + m * (2n+1) = 3mn + m + n$ failures. As an example, a 25/5 system can tolerate at least 8 and up to 16 failures out of 25 total replicas. A 21/3 system can tolerate at least 7, and a maximum of 12, failures out of 21 total replicas. Individual subquorums might still be able to perform local operations despite an impasse at the global level.

Total subquorum failure can temporarily cause a portion of the namespace to be unserved. However, the root quorum eventually times out and moves into a new epoch with that portion assigned to another subquorum.

3.4.2 Obligations Timeout

write this section

3.4.3 The Nuclear Option

fix this section

Singleton consensus protocols, including Raft, can tolerate just under half of the entire system failing. As described above, HC's structure makes it more vulnerable to clustered failures. Therefore we define a *nuclear option*, which uses direct consensus decision among all system replicas to tolerate any f replicas failing out of $2f + 1$ total replicas in the system.

A nuclear vote is triggered by the failure of a root leader election. A *nuclear candidate* increment's its term for the root quorum and broadcasts a request for votes to all system replicas. The key difficulty is in preventing delegated votes

and nuclear votes from reaching conflicting decisions. Such situations might occur when temporarily unavailable subquorum leaders regain connectivity and allow a wedged root quorum to unblock. Meanwhile, a nuclear vote might be concurrently underway.

Replica delegations are defined as intervals over specific slots. Using local subquorum slots would fall prey to the above problem, so we define delegations as a small number (often one) of root slots, which usually correspond to distinct epochs. During failure-free operation, peers delegate to their leaders and are all represented in the next root election or commit. Peers then renew their delegations to their leaders by appending them to the next local commit reply. This approach works for replicas that change subquorums over an epoch boundary, and even allows peers to delegate their votes to arbitrary other peers in the system (see replicas r_N and r_O in Figure 3.3).

This approach is simple and correct, but deals poorly with leader turnovers in the subquorum. Consider a subquorum where all peers have delegated votes to their leader for the next root slot. If that leader fails, none of the peers will be represented. We finesse this issue by re-defining such delegations to count root elections, root commits, *and* root heartbeats. The latter means that local peers will regain their votes for the next root quorum action if it happens after to the next heartbeat.

Consider the worst-case failure situation discussed in Section 3.4: a majority of the majority of subquorums have failed. None of the failed subquorum leaders can be replaced, as none of those subquorums have enough local peers.

The first response is initiated when a replica holding delegations (or its own vote) times out waiting for the root heartbeat. That replica increments its own root term, adopts the prior system configuration as its own, and becomes a root candidate. This candidacy fails, as a majority of subquorum leaders, with all of their delegated votes, are gone. Progress is not made until delegations time out. In our default case where a delegation is for a single root event, this happens after the first root election failure.

At the next timeout, any replica might become a candidate because delegations have lapsed (under our default assumptions above). Such a *nuclear* candidate increments its root term and sends candidate requests to all system replicas, succeeding if it gathers a majority across all live replicas.

The first candidacy assumed the prior system configuration in its candidacy announcement. This configuration is no longer appropriate unless some of the "failed" replicas quickly regain connectivity. Before the replica announces its candidacy for a second time, however, many of the replica replies have timed out. The candidate alters its second proposed configuration by recasting all such replicas as hot spares and potentially reducing the number and size of the subgroups. Subsequent epoch changes might re-integrate the new hot spares if the replicas regain connectivity.

3.5 Performance Evaluation

HC was designed to adapt both to dynamic workloads as well as variable network conditions. We therefore evaluate HC in two distinct environments: a

homogeneous data center and a heterogeneous real-world network. The homogeneous cluster is hosted on Amazon EC2 and includes 26 "t2.medium" instances: dual-core virtual machines running in a single VPC with inter-machine latencies of $\lambda_\mu = 0.399ms$ and $\lambda_\sigma = 0.216ms$. These machines are cost effective and, though lightweight, are easy to scale to large cluster sizes as workload increases. Experiments are set up such that each instance runs a single replica process and multiple client processes.

The heterogeneous cluster (UMD) consists of several local machines distributed across a wide area, with inter-machine latencies ranging from $\lambda_\mu = 2.527ms$, $\lambda_\sigma = 1.147ms$ to $\lambda_\mu = 34.651ms$, $\lambda_\sigma = 37.915ms$. Machines in this network are a variety of dual and quad core desktop servers that are solely dedicated to running these benchmarks. Experiments on these machines are set up so that each instance runs multiple replica and client processes co-located on the same host. In this environment, localization is critical both for performance but also to ensure that the protocol can elect and maintain consensus leadership. The variability of this network also poses challenges that HC is uniquely suited to handle via root quorum-guided adaptation. We explore two distinct scenarios – sawtooth and repartitioning – using this cluster; all other experiments were run on the EC2 cluster.

HC is partially motivated by the need to scale strong consistency to large cluster sizes. We based our work on the assumption that consensus performance decreases as the quorum size increases, which we confirm empirically in Figure 3.9. This figure shows the maximum throughput against system size for a variety of workloads, up to 120 concurrent clients. A workload consists of one or more clients

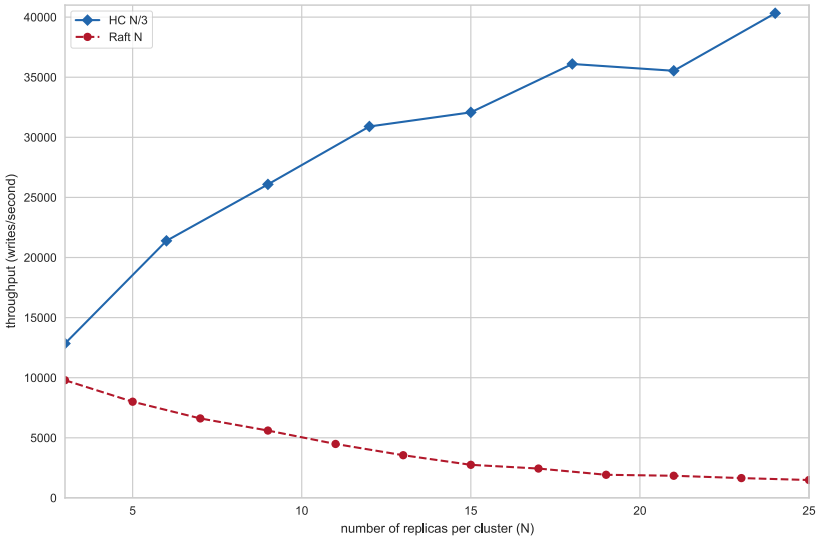

Figure 3.9: Mean throughput of workloads of up to 120 concurrent clients

continuously sending writes of a specific object or objects to the cluster without pause.

Standard consensus algorithms, Raft in particular, scale poorly with uniformly decreasing throughput as nodes are added to the cluster. Commit latency increases with quorum size as the system has to wait for more responses from peers, thereby decreasing overall throughput. Figures 3.9 and 3.10 clearly show the multiplicative advantage of HC's hierarchical structure, though HC does not scale linearly as we had expected.

There are at least two factors currently limiting the HC throughput shown here. First, the HC subquorums for the larger system sizes are not saturated. A single 3-node subquorum saturates at around 25 clients and this experiment has only

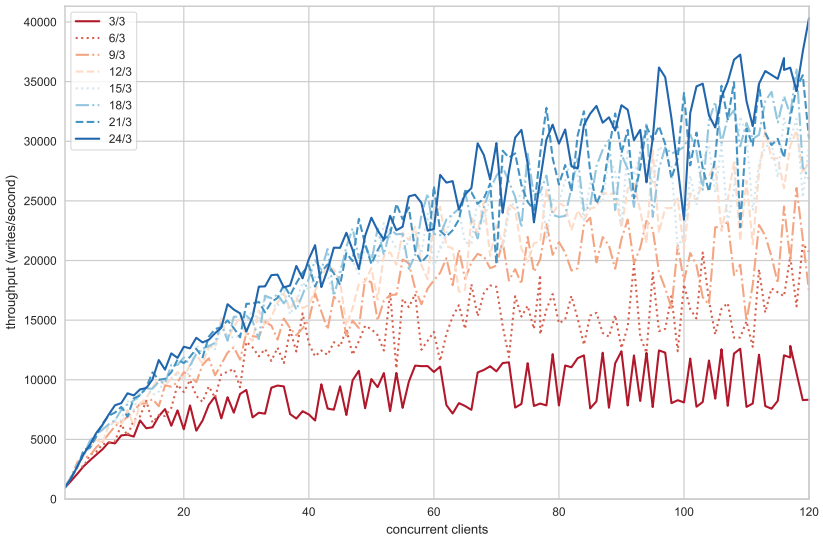

Figure 3.10: Performance of distributed consensus with an increasing workload of concurrent clients. Performance is measured by throughput, the number of writes committed per second.

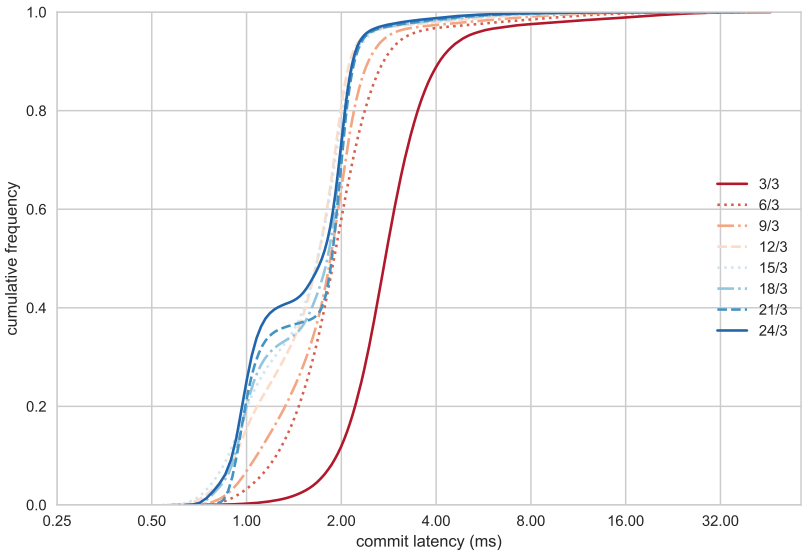

Figure 3.11:

about 15 clients per subquorum for the largest cluster size. We ran experiments with 600 clients, saturating all subquorums even in the 24-node case. This throughput peaked at slightly over 50,000 committed writes per second, better but still lower than the linear scaling we had expected.

We think the reason for this ceiling is hinted at by Figure 3.10. This figure shows increasingly larger variability with increasing system sizes. A more thorough examination of the data shows widely varying performance across individual subquorums in the larger configurations. We suspect that the cause is either VM misconfiguration or misbehavior. We are adding more instrumentation to diagnose the problem.

The effect of saturation is also demonstrated in Figure 3.11, which shows cumu-

lative latency distributions for different system sizes holding the workload (number of concurrent clients) constant. The fastest (24/3) shows nearly 80% of client write requests being serviced in under 2 msec. Larger system sizes are faster because the smaller systems suffer from contention (25 clients can saturate a single subquorum). Because throughput is directly related to commit latency, throughput variability can be mitigated by adding additional subquorums to balance load.

Besides pure performance and scaling, HC is also motivated by the need to adapt to varying environmental conditions. In the next set of experiments, we explore two common runtime scenarios that motivate adaptation: shifting client workloads and failures. We show that HC is able to adapt and recover with little loss in performance. These scenarios are shown in Figures 3.12 and 3.13 as throughput over time, where vertical dotted lines indicate an epoch change.

The first scenario, described by the time series in Figure 3.12 shows an HC 3-replica configuration moving through two epoch changes. Each epoch change is triggered by the need to localize tags accessed by clients to nearby subquorums. The scenario shown starts with all clients co-located with the subquorum serving the tag they are accessing. However, clients incrementally change their access patterns first to a tag located on one remote subquorum, and then to the tag owned by the other. In both cases, the root quorum adapts the system by repartitioning the tagspace such that the tag defining their current focus is served by the co-located subquorum.

Finally, Figure 3.13 shows a 3-subquorum configuration where one entire subquorum becomes partitioned from the others. After a timeout, the root uses an epoch change to re-allocate the tag of the partitioned subquorum over the two

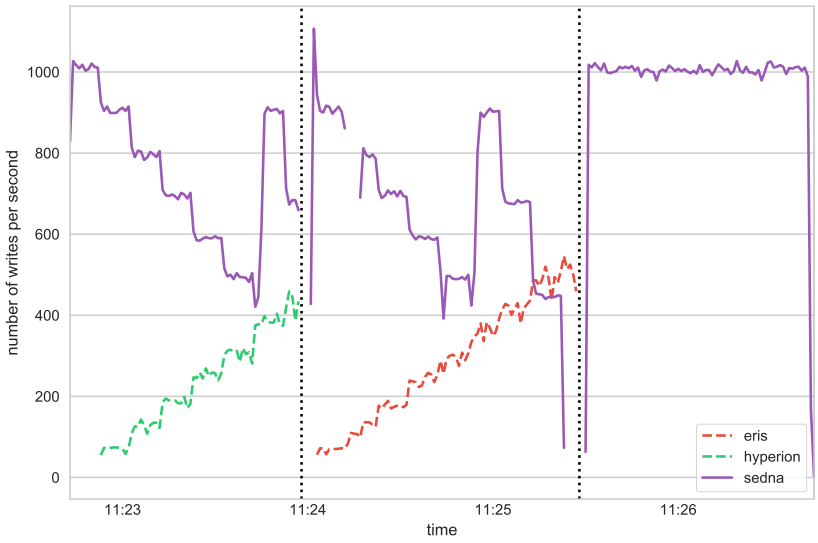

Figure 3.12: 9/3 system adapting to changing client access patterns by repartitioning the tag space so that clients are co-located with subquorums that serve tags they need.

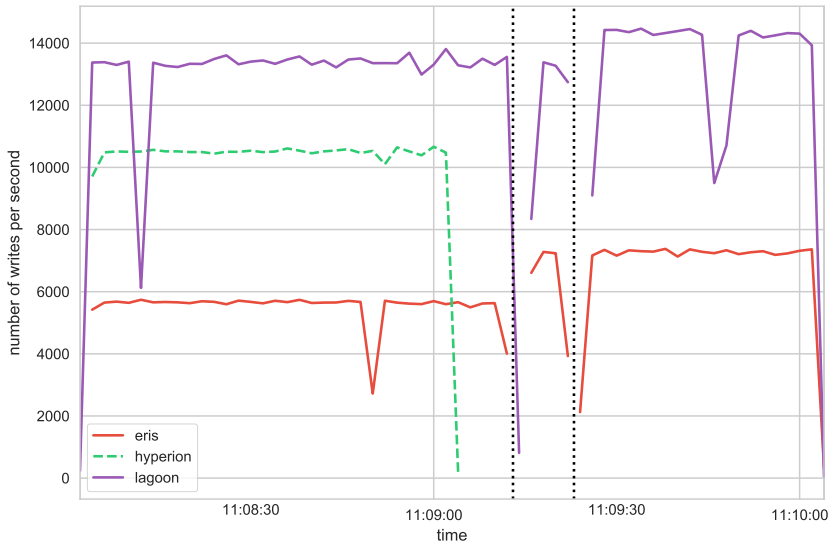

Figure 3.13: 9/3 System that adapts to failure (partition) of entire subquorum. After timeout, the root quorum re-partitions the tag allocated to the failed subquorum among the other two subquorums.

remaining subquorums. The partitioned subquorum eventually has an *obligation timeout*, after which the root quorum is not obliged to leave the tag with the current subquorum. The tag may then be re-assigned to any other subquorum. Timeouts are structured such that by the time an obligation timeout fires, the root quorum has already re-mapped that subquorum's tag to other subquorums. As a result, the system is able to recover from the partition as fast as possible. Note that in this figure, the repartition occurs through two epoch changes, the first allocating part of the tagspace to the first subquorum, and the second allocating the rest of the tag to the other. Gaps in the graph are periods where the subquorums are electing local leaders. This may be optimized by having leadership assigned or maintained through root consensus.

3.6 Conclusion

Most consensus algorithms have their roots in the Paxos algorithm, originally described in parliamentary terms. The metaphor of government still applies well as we look at the evolution of distributed coordination as systems have grown to include large numbers of processes and geographies. Systems that use a dedicated leader are easy to reason about and implement, however, like chess, if the leader goes down the system cannot make any progress. Simple democracies for small groups solve this problem but do not scale, and as the system grows, it fragments into tribes. Inspired by modern governments, we have proposed a representative system of consensus, hierarchical consensus, such that replicas elect leaders to participate

in a root quorum that makes decisions about the global state of the system. Local decision making, the kind that effects only a subset of clients and objects is handled locally by subquorums as efficiently as possible. The result is a a hierarchy of decision making that takes advantage of hierarchies that already exist in applications.

Hierarchical Consensus is an implementation and extension of Vertical Paxos. Like Vertical Paxos, HC reasons about consistency across all objects by identifying commands with a grid ordering (rather than a log ordering) and is reconfigurable to adapt to dynamic environments that exist in geo-replicated systems. Adaptability allows HC to exploit locality of access, allowing for high performance coordination, even with replication across the wide area. HC extends Vertical Paxos to ensure that intersections exist between the subquorums and the root quorum to ensure coordination exists between subquorums and to ensure that the system operates as a coordinated whole. To scale the consensus protocol of the root quorum, we propose a novel approach, delegation, to ensure that all replicas participate in consensus but limit the number and frequency of messages required to achieve majority. Finally, we generalized HC from primary-backup replication to describe more general online replication required by distributed databases and file systems.

In the next chapter we'll explore a hybrid consistency model implemented by federating replicas that participate in different consistency protocols. In a planetary scale network, HC provides the strong consistency backbone of the federated model, increasing the overall consistency of the system by making coordinating decisions at a high level, and allowing high availability replicas in the fog operate independently where necessary.

Chapter 4: Federated Consistency

Hybrid consistency model

4.1 Overview

4.2 Eventual Consistency

Read/Write Quorums

Clients can `Put` (write) and `Get` (read) key-value pairs to and from one or more replicas in a single operation. The set of replicas that responds to a client creates a quorum that must agree on the state of the operation at its conclusion. Clients can vary read and write quorum sizes to improve consistency or availability – larger quorums reduce the likelihood of inconsistencies caused by concurrent updates, but smaller quorums respond much more quickly, particularly if the replicas in the quorum are co-located with the client. In large, geo-replicated systems we assume that clients will prefer to choose fewer, local replicas to connect with, optimistic that collisions across the wide-area are rare, e.g. that writes are localized but reads are global.

On `Put`, the instance of the key-value pair created by the update is assigned

a monotonically increasing, conflict-free *version number* [66, 67]. For simplicity, we assume a fixed number of replicas, therefore each version is made up of two components: the *update* and *precedence ids*. Precedence ids are assigned to replicas during configuration, and update ids are incremented to the largest observed value during synchronization. As a result, any two versions generated by a `Put` anywhere in the system are comparable such that the *latest* version of the key-value pair is the version with the largest update id, and in the case of ties, the largest precedence id.

Additional version metadata, including the parent version of the update (in a read-then-write system or simply the latest version of the key stored locally), implements a virtual object history that allows us to reason about consistency. Keys can be managed independently, e.g. each key has its own update id sequence resulting in per-object consistency, or all objects can be managed together with a single sequence; in the latter case, it is possible to construct an ordering history of operations to all objects and in the former, a sequence of operations for each object. Object histories allow us to reason about the global consistency of the system.

There are two primary inconsistencies that can occur in this system: *stale reads* and *forked writes*. A stale read means that the `Get` operation has not returned the globally most recent version of the object, e.g. the local replica is behind in the object history. A forked write is caused when there are two concurrent writes to the same object, a symptom of stale reads. Forked writes cause a divergence in the object history such that there are two or more branches of update operations. As we will see in the next section, one of these writes will eventually be *stomped* before

it can become fully replicated, meaning that the eventual consistency prunes these branches at the cost of losing the update. The ideal consistency for a system is represented by a linear object history without forks [68], which demonstrates that the system was in a consistent state during all accesses.

Anti-Entropy Sessions and Synchronization

Anti-entropy sessions are conducted in a pairwise fashion on a periodic interval to ensure that the network is not saturated with synchronization requests which may reduce client availability. At each interval, every replica selects a synchronization partner such that all replicas have a uniform likelihood of selection. This ensures that an update originating at one replica will be propagated to all online replicas given the continued operation of replication. This mechanism also provides robustness in the face of failure; a single unresponsive replica or even network partition does not become a bottleneck to synchronization, and once the failure is repaired synchronization will occur without reconfiguration.

There are two basic forms of synchronization: *push* synchronization is a fire-and-forget form of synchronization where the remote replica is sent the latest version of all objects, whereas *pull* synchronization requests the latest version of objects and minimizes the size of data transfer. To get the benefit of both, we consider *bilateral* synchronization which combines push and pull in a two-phase exchange. Bilateral synchronization increases the effect of anti-entropy during each exchange because it ensures that in the common case each replica is synchronized with two other replicas instead of one during every anti-entropy period.

Bilateral anti-entropy starts with the initiating replica sending a vector of the

latest local versions of all keys currently stored, usually optimized with Merkel or prefix trees to make comparisons faster. The remote replica compares the versions sent by the initiating replica with its current state and responds with any objects whose version is *later* than the initiating replica's as well as another version vector of requested objects that are earlier on the remote. The initiating replica then replies with the remote's requested objects, completing the synchronization. We refer to the first stage of requesting later objects from the remote as the pull phase, and the second stage of responding to the remote the push phase.

There are two important things to note about this form of anti-entropy exchange. First, this type of synchronization implements a *latest writer wins* policy. This means that not all versions are guaranteed to become fully replicated – if a later version is written during propagation of an earlier version, then the earlier version gets *stomped* by the later version because only the latest versions of objects are exchanged. If there are two concurrent writes, only one write will become fully replicated, the write on the replica with the greater precedence.

Policies: latest writer wins

Bilateral anti-entropy

4.2.1 Consistency Failures

Forks

Stale Reads

4.3 Integration

4.3.1 Communication Integration

4.3.2 Consistency Integration

- Forte Number

4.4 Performance Evaluation

Communication Topology Inconsistencies due to outages Inconsistency due to system latency

Chapter 5: System Implementation

Given its grandiose title, it may seem that the engineering behind the development of a planetary scale data storage system would require thousands of man-hours of professional software engineers and a highly structured development process. In fact, this is not necessarily the case for two reasons. First, data systems benefit from an existing global network topology and commercial frameworks for deploying applications. This means that both the foundation and motivation for creating large geo-replicated systems exists, as described earlier. Second, like the internet, complex global systems emerge through the composition of many simpler components following straight forward rules [69]. Instead of architecting a monolithic system, the design process is decomposed to reasoning about the behavior of single processes. Rather than being built, a robust planetary data system evolves from its network environment.

To facilitate system evolution, the consistency models we have described thus far have been *composable* to allow heterogeneous replicas to participate in the same system. However, while composability allows us to reason about consistency expectations, it does not necessarily mean interoperability. In order to ensure reason about system expectations we must outline our assumptions for communication, se-

curity, processing, and data storage. In this chapter, we describe the implementation of the replicas and applications of our experimental system and the assumptions we made.

5.1 System Model

A *replica* is an independent process that maintains a portion of the objects stored as well as a *view* of the state of the entire system. Replicas must be able to communicate with one another and may also *serve* requests from clients. A system is composed of multiple communicating replicas and is defined by the behavior the replicas. For example, a totally replicated system is one where each replica stores a complete copy of all objects as in the primary-backup approach [70], whereas a partially replicated system ensures durability such that multiple replicas store the same object but not all replicas store all objects as in the Google File System [71]. At the scale of a multi-region, globally deployed system, we assume that total replication is impractical and primarily consider the partial replication case.

Let's unpack some of the assumptions made by the seemingly simple statements made in the previous paragraph. First, independence means replicas have a shared-nothing architecture [72] and cannot share either memory or disk space. For practical purposes of fault tolerance, we generally assume that there is a one-to-one relationship between a replica and a disk so that a disk failure means only a single replica failure. Second, that each replica must maintain a view of the state of the entire system means both that replicas must be aware of their peers on the network

and that they should know the locations of objects stored on the network. A strict interpretation of this requirement would necessarily make system membership brittle as it would be difficult to add or remove replicas. Alternatively, a centralized interpretation of the view requirement would allow for

Second, the ability to communicate with replicas and serve requests from clients means that the replica must be addressable.

This requirement is seemingly innocuous when taken by itself, however when we also describe a replica as requiring a view of the state of the entire system, it means that a replica must know about the existence of all other replicas in the system. A strict interpretation of having a complete view would necessarily make the system composition brittle, unable to add or remove replicas. Instead we take a less strict view

- networking - actors - event loop - reasons why the above are important -

5.2 Applications

5.2.1 Distributed Log

5.2.2 Key-Value Database

5.2.3 File System

5.3 Consistency Model

Client-side vs. system-side consistency

Log model of consistency

Continuous consistency scale

Grid consistency model

5.4 Raft

Hierarchical Consensus with modified Raft as the underlying consensus protocol exports a linearizable order of accesses to the distributed key-value store. Alia and the HC library are implemented in Golang use gRPC [73] for communication. The system is implemented in 7,924 lines of code, not including standard libraries or support packages.

An replica implements multiple instantiations of the Raft protocol, which we have modified in several ways. Every replica must run one instantiation of the *root consensus protocol*. Replicas may also run one or more instantiations of the *commit consensus protocol* if they are assigned to a subquorum. Repartition decisions move the system between epochs with a new configuration and tagspace, and can only be initiated by messages from peers or monitoring processes. A successful repartition results in a new epoch, tagspace, and subquorum topology committed to the root log. `Repartition` messages also serve to notify the network about events that do not require an epoch change, such as the election of a new subquorum leader or bringing a failed node back online.

Each replica implements an event loop that responds to timing events, client requests, and messages from peers. Events may cause the replica to change state,

modify a command log, broadcast messages to peers, modify the key-value store, or respond to a client. Event handlers need to aggressively lock shared state for correctness because Golang and gRPC make extensive use of multi-threading. The balance between correctness and concurrency-driven performance leads to increasing complexity and tighter coupling between components, one that foreshadows extra-process consistency concerns that have been noted in other work [49, 74, 75].

The computing and network environment of a distributed system plays a large role in determining not just the performance of the system, but also its behavior. A simple example is the election timeout parameter of the Raft consensus protocol, which must be much greater than the average time to broadcast and receive responses, and much less than the mean time between failures [49, 76, 77]. If this requirement is not met, leader may be displaced before heartbeat messages arrive, or the system will be unable to recover when a leader fails. As a result, the relationship between timeouts is critically dependent on the mean latency (λ_μ) of the network. Howard [78] proposes $T = \lambda_\mu + 2\lambda_\sigma$ to determine timeouts based on the distribution of observed latencies, sets the heartbeat as $\frac{T}{2}$, and the election timeout as the interval $U(T, 2T)$. We parameterize our timeouts (Table 5.1) on latency measurements made before we ran our experiments. Monitoring and adapting to network conditions is part of ongoing work.

Changes to base Raft: In addition to major changes, such allowing replicas to be part of multiple quorums simultaneously, we also made many smaller changes that had pervasive effects. One change was including the *epoch* number alongside the term in all log entries. The epoch is evaluated for invariants such as whether or

Table 5.1: Parameterized timeouts in our implementation. The *obligation* timeout stops a partitioned subquorum after an extended time without contact to the rest of the system. $T = 10msec$ for our experiments on Amazon EC2.

Name	Time	Actions
sub heartbeat sub leader	1T 2-4T	sub leader heartbeat new sub election
root heartbeat root election	10T 20-40T	root leader heartbeat new root election
obligation	50T	root quorum may re-allocate the tag

not a replica can append an entry or if a log is as up to date as a remote log.

Vote delegation requires changes to vote counting. Since our root quorum membership actually consists of the entire system, all replicas are messaged during root events. All replicas reply, though most with a "zero votes" acknowledgment. The root uses observed vote distributions to inform the ordering of future consensus messages (sending requests first to replicas with votes to cast), and uses timeouts to move non-responsive replicas into "hot spares" status.

We allow `AppendEntries` requests in subquorums to aggregate multiple client requests into a single consensus round. Such requests are collected while an outstanding commit round is ongoing, then sent together when that round completes. The root quorum also aggregates all requests within a minimum interval into a single new epoch-change/reconfiguration operation to minimize disruption.

Commits are observed by the leader once a majority of replicas respond positively. Other replicas learn about the commit only on the next message or heartbeat. Root epoch changes and heartbeats are designed to be rare, meaning that epoch

change commits are not seen promptly. We modified the root protocol to inform subquorums of the change by sending an additional heartbeat immediately after it observes a commit.

Replicas may be part of both a subquorum and the root quorum, and across epoch boundaries may be part of multiple subquorums. In principle, a high performance replica may participate in any number of subquorums. We therefore allow replicas to accommodate multiple distinct logs with different access characteristics.

Peers that are either slow or with unsteady connectivity are occasionally left behind at subquorum leader or epoch changes. Root heartbeats containing the current system configuration are broadcast to all replicas and serve to bring them up to date.

Finally, consensus protocols often synchronously write state to disk before responding to remote requests. This allows replicas that merely crash to reboot and rejoin the ongoing computation after recovering state from disk. Otherwise, these replicas need to go through heavyweight leave-and-rejoin handshakes. Our system avoids these synchronous writes by allowing epochs to re-join a subquorum at the next epoch change without any saved state, avoiding these handshakes altogether.

5.5 Conclusion

We did not optimize our research for the minimum set of assumptions required to facilitate interoperability between heterogeneous replicas. However, we hope that the assumptions we did make shed light on what is required to achieve the minimum

set of assumptions.

Further research is required to ...

Chapter 6: Adaptive Consistency

Throughout this dissertation we've outlined a planetary-scale data storage system composed of a two-tier structure that provides a hybrid consistency model. Both tiers are designed to scale to thousands of replicas, and together could represent millions of replicas operating in concert around the world. Management and systems administration of such a large scale system using external monitoring processes is impractical at best and prohibitively complex at worse. Even with a trusted infrastructure of cloud services, building a single synchronization point for monitoring and optimization would require the online collection of live information from across the globe, which itself would be susceptible to delays and partitions of the kind the administrator would be trying to manage. Even if those delays could be managed, such a synchronization point would have to manage a huge number of events streaming in from a large number of sources, which has challenges in and of itself [79].

Instead, we propose that an *emergent model* of network behavior is required to tune and optimize planetary scale systems in an online fashion such that local, simple rules lead to globally emergent behavior [80]. Specifically we hypothesize that when individual replicas follow simple optimization procedures based on monitoring

of their local network performance, access patterns, queries to their neighbors, and other environmental factors the performance of the system will collectively increase. Because we focus primarily on the consistency aspects of geo-replicated data storage, we have termed this behavior *adaptive consistency*, because with a hybrid or continuous consistency model, such optimizations will minimize inconsistent behaviors due to latency or configuration.

Although this work is largely left for future research on a fully deployed platform, we have built our system with this kind of adaptation in mind. To validate our thought process, we have conducted an experiment that primarily adapts the federated fog layer of our system by modifying anti-entropy selection with reinforcement learning techniques [81]. In this chapter we will show we can improve consistency of the system as a whole with localized machine learning implemented on a per-replica basis. We will then finish with a discussion of how we can generalize this process to other techniques in the system as a whole.

6.1 Anti-Entropy Bandits

A distributed system is made highly available when individual servers are allowed to operate independently without failure-prone, high latency coordination. The independent nature of the server's behavior means that it can immediately respond to client requests, but that it does so from a limited, local perspective which may be inconsistent with another server's response. If individual servers in a system were allowed to remain wholly independent, individual requests from clients

to different servers would create a lack of order or predictability, a gradual decline into inconsistency, i.e. the system would experience *entropy*. To combat the effect of entropy while still remaining highly available, servers engage in periodic background *anti-entropy sessions* [60].

Anti-entropy sessions synchronize the state between servers ensuring that, at least briefly, the local state is consistent with a portion of the global state of the system. If all servers engage in anti-entropy sessions, the system is able to make some reasonable guarantees about consistent replication; the most famous of which is that without requests the system will become globally consistent, eventually [61]. More specifically, inconsistencies in the form of stale reads can be bound by likelihoods that are informed by the latency of anti-entropy sessions and the size of the system [82, 83]. Said another way, overall consistency is improved in an eventually consistent system by decreasing the likelihood of a stale read, which is tuned by improving the *visibility latency* of a write, the speed at which a write is propagated to a significant portion of servers. This idea has led many system designers to decide that eventual consistency is "consistent enough" [84, 85], particularly in a data center context where visibility latency is far below the rate of client requests, leading to practically strong consistency.

However, propagation rates need to be re-evaluated when replicas move outside of data center contexts and when anti-entropy is replicating across the wide area. Our system envisions a fog layer that provides data services to localized regions with a hybrid consistency model. The edge is specifically designed to handle mobile users, sensor systems, and high throughput applications the edge of the

data center backbone [86, 87]. However, scaling an eventually consistent system to dozens or even hundreds of nodes increases the radius of the network, which leads to increased noise during anti-entropy e.g. the possibility that an anti-entropy session will be between two already synchronized nodes. Geographic distribution and extra-datacenter networks also increase the latency of anti-entropy sessions so that inconsistencies become more apparent to external observers.

To address this challenge, we propose the use of reinforcement learning techniques to optimize network behavior to minimize latency. Anti-entropy uses gossip and rumor spreading to propagate updates deterministically without saturating the network even in the face of network outages [88–90]. These protocols use uniform random selection to choose synchronization peers, which means that a write occurring at one replica is not efficiently propagated across the network. In this section we explore the use of *multi-armed bandit* algorithms [91, 92] to optimize for fast, successful synchronizations by modifying peer selection probabilities. The result is a synchronization topology that emerges according to access patterns and network latencies. Such topologies produce efficient synchronization, localize most data exchanges, lower visibility latency, and increase consistency.

6.1.1 Accesses and Consistency

In this section we review the access and consistency model in the context of bandits as well as how anti-entropy is conducted. A more complete discussion of these topics can be found in Chapter 4.

Clients can `Put` (write) and `Get` (read) key-value pairs to and from one or more replicas in a single operation, creating read and write quorums that improve consistency by enforcing coordination between replicas on the access. In large, geo-replicated systems, we assume that clients prefer to choose fewer, local replicas to connect with, assuming that writes are primarily local and reads are global. On `Put`, a new conflict-free version of the write is created. This results in the possibility of two types of inconsistencies that occur during concurrent accesses: stale reads and forked writes. As a write is propagated through the system, the latest-writer wins policy means that at least one of the forks will be "stomped", e.g. not fully replicated.

Both forms of inconsistency can be primarily attributed to *visibility latency*, that is the time it takes for an update to propagate to all replicas in the system. Visibility latency is directly related to the likelihood of stale reads with respect to the frequency of accesses [83]; said another way, decreasing the visibility latency improves the overall consistency of a system. However, in a system that uses anti-entropy for replication, the propagation speed of an update is not governed solely by network connections, it is also bound to the number and frequency of anti-entropy sessions conducted as well as the radius of the network.

Visibility latency is minimized when all replicas choose a remote synchronization partner that does not yet have the update. This means that minimal visibility latency is equal to $t \log_3 n$, where t is the anti-entropy interval and n is the number of replicas in the network. In practice, however, because of inefficient exchanges due to uniform random selection of synchronization partners, this latency is never prac-

tically achieved, and is instead modulated by a noise variable that is proportional to the size of the network.

6.1.2 Multi-Armed Bandits

To combat the effect of noise on visibility latency our initial approach employs a technique commonly used in active and reinforcement learning: multi-armed bandits. Multi-armed bandits refer to a statistical optimization procedure that is designed to find the optimal payout of several choices that each have different probabilities of reward. In this case, we use bandits to improve uniform random selection of peers so that replicas choose synchronization partners that are most likely to exchange information, and thus more quickly propagate updates, while still maintaining the properties of full replication and fault tolerance.

A bandit problem is designed by identifying several (usually more than two) competing choices called "arms" [1] , as well as a reward function that determines how successful the selection of an arm is. During operation, the bandit selects an arm, observes the rewards, then updates the payout likelihood of the selected arm, normalized by the number of selections. As the bandit selects arms, it learns which arm or arms have the highest likelihood of reward, and can modify it's arm selection *strategy* to maximize the total reward over time.

Bandits must balance exploration of new arms with possibly better reward values and exploitation of an arm that has higher rewards than the other. In the

[1] Arms refer to the pulling mechanism of a slot machine, the metaphor generally used to motivate the multi-armed bandit problem.

epsilon greedy strategy, the bandit will select the arm with the best reward with some probability $1 - \epsilon$, otherwise it will select any of the arms with uniform probability. The smaller ϵ is, the more the bandit favors exploitation of known good arms, the larger ϵ is, the more it favors exploration. If $\epsilon = 1$ then the algorithm is simply uniform random selection. A simple extension of this is a strategy called *annealing epsilon greedy*, which starts with a large ϵ, then as the number of trials increases, steadily decreases ϵ on a logarithmic scale. There are many other bandit strategies but we have chosen these two simple strategies for our initial research to demonstrate a bolt-on effective improvement to existing systems.

Peer selection for anti-entropy is usually conducted with uniform random selection to guarantee complete replication. To extend anti-entropy with bandits, we design a selection method whose arms are remote peers and whose rewards are determined by the success of synchronization. The goal of adding bandits to anti-entropy is to optimize selection of peers such that the visibility latency becomes closer to the optimal propagation time as a synchronization topology emerges from the bandits. A secondary goal is to minimize anti-entropy latency by preferring local (in the same data center) and regional (e.g. on the same continent) connections.

Our initial reward function favors synchronizations to replicas where the most writes are occurring by giving higher rewards to anti-entropy sessions that exchange later versions in either a push or a pull, as well as additional rewards if more than one object is exchanged. Additionally, the latency of the synchronization RPCs is computed to reward replicas that are near each other. The complete reward function is given in Table 6.1: for each phase of synchronization (push and pull), compute the

reward as the sum of the propositions given. For example if a synchronization results in three objects being pulled in 250ms, and one object being pushed in 250ms, the reward is 0.75.

Table 6.1: The rewards function for our initial anti-entropy bandits. Rewards are computed by introspecting the results of the pull and push phases of bilateral anti-entropy.

	Pull	**Push**	**Total**
Synchronize at least 1 object	0.25	0.25	0.50
Additional for multiple objects	0.05	0.05	0.10
Latency $\leq$ 5ms (local)	0.10	0.10	0.20
Latency $\leq$ 100ms (regional)	0.10	0.10	0.20
Total	*0.50*	*0.50*	*1.00*

The design of reward functions can be implemented to the needs of a specific system. For example, in a system that has workloads with variable sized writes, object size could be considered or systems with imbalanced deployments might consider a reward function that prioritizes inter-region communication.

6.1.3 Experiments

We conducted experiments using a distributed key-value store totally replicated across 45 replicas in 15 geographic regions on 5 continents around the world. Replicas were hosted using AWS EC2 t2.micro instances and were connected to each other via internal VPCs when in the same region, using external connections between regions. The store, called Honu, is implemented in Go 1.9 using gRPC and protocol buffers for RPC requests; all code is open source and available on GitHub.

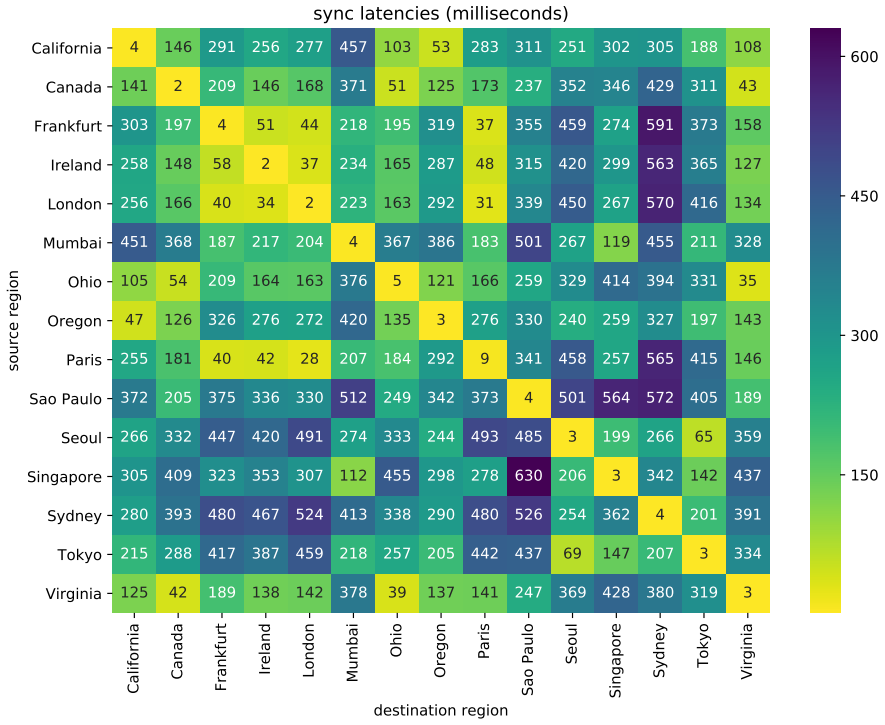

Figure 6.1: Inter-Region Synchronization Latencies (Push+Pull)

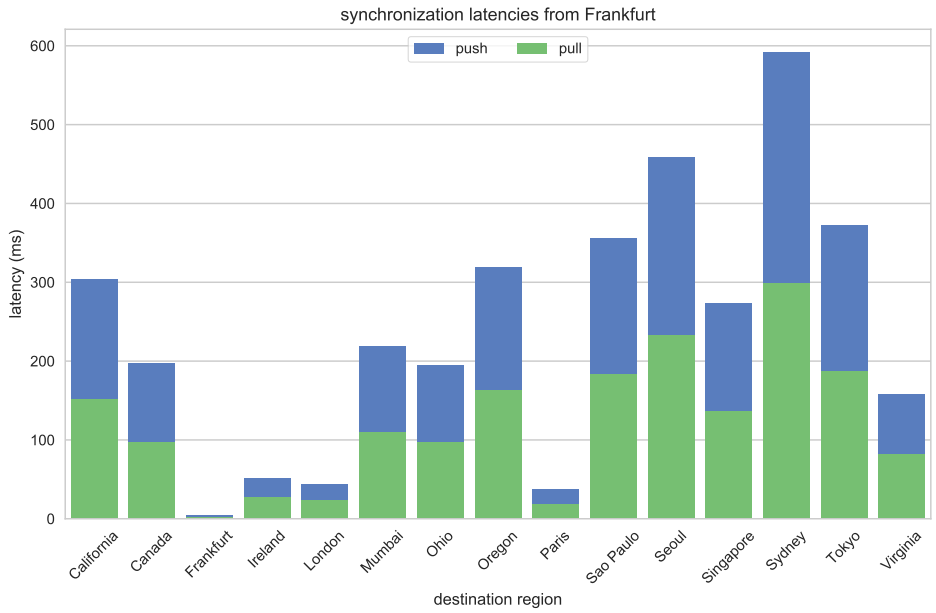

Figure 6.2: View of anti-entropy synchronization latency from Europe and corresponding network distances.

The workload on the system was generated by 15 clients, one in each region and colocated with one of the replicas. Clients continuously created Put requests for random keys with a unique prefix per-region such that consistency conflicts only occur within a single region. The average throughput generated per-client was 5620.4 puts/second. The mean synchronization latency between each region ranged from 35ms to 630ms as shown in Figures 6.1 and 6.2. To ensure at least one synchronization per anti-entropy session, we set the anti-entropy interval to 1 second to train the system, then reduced the interval to 125ms while measuring visibility latency. To account for lag between commands sent to replicas in different regions, each experiment was run for 11 minutes, the bandit learning period was 4 minutes then visibility latency was observed for 6 minutes, buffered by 30 seconds

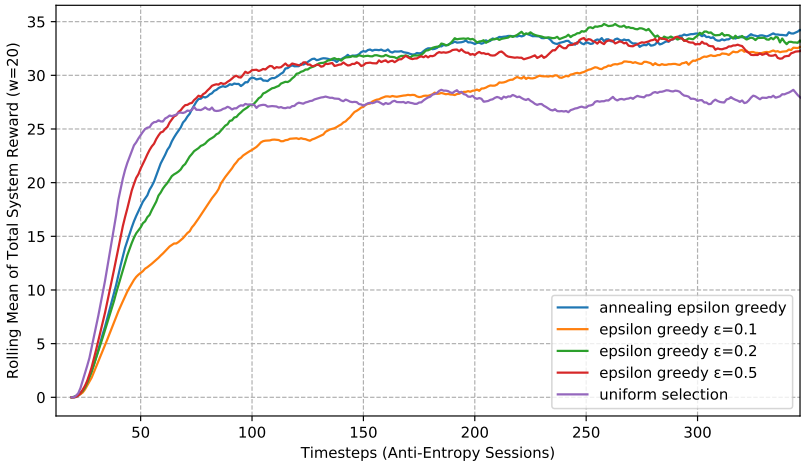

Figure 6.3: Total system rewards over time.

before and after the workload to allow replicas to initialize and gracefully shutdown.

Our first experiments compared uniform random peer selection with epsilon greedy bandits using $\epsilon \in \{0.1, 0.2, 0.5\}$ as well as an annealing epsilon greedy bandit. The total system rewards as a rolling mean over a time window of 20 synchronizations are shown in Figure 6.3. The rewards ramp up from zero as the clients come online and start creating work to be synchronized. All of the bandit algorithms eventually improve over the baseline of uniform selection, not only generating more total reward across the system, but also introducing less variability in rewards over time. None of the bandit curves immediately produces high rewards as they explore the reward space; lower ϵ values may cause exploitation of incorrect arms, while higher ϵ values take longer to find optimal topologies. However, in the static workload case, the more aggressive bandit strategies converge more quickly to the optimal reward.

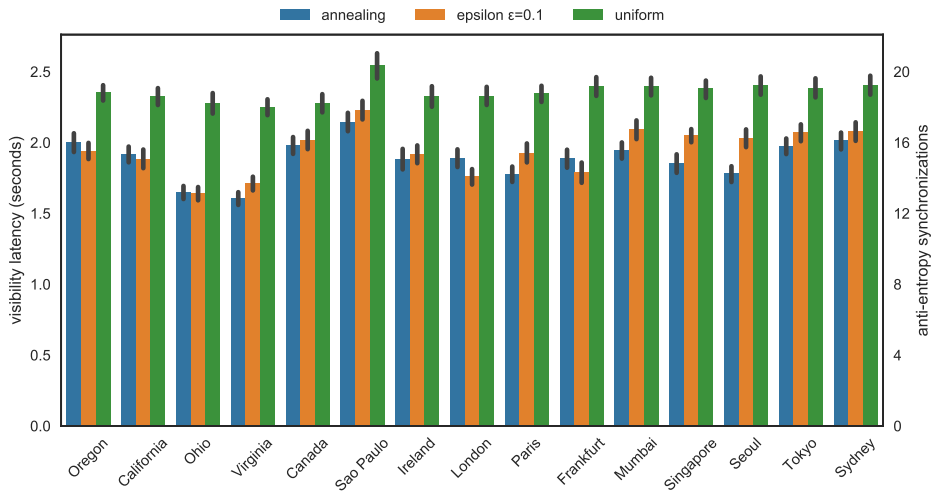

Figure 6.4: Decreasing visibility latency from bandit approaches.

Visibility latencies were computed by reducing the workload rate to once every 4 seconds to ensure the write becomes fully visible across the entire network. During the visibility measurement period, replicas locally logged the timestamp the write was pushed or pulled; visibility latency is computed as the difference between the minimum and maximum timestamp. The average visibility latency per region is shown in Figure 6.4 measured by the left y-axis. Because the anti-entropy delay is a fixed interval, the estimated number of required anti-entropy sessions associated with the visibility delay is shown on the right y-axis of the same figure. Employing bandit strategies reduces the visibility latency from 2360ms on average in the uniform case to 1870ms, reducing the number of required anti-entropy intervals by approximately 4.

To show the emergent behavior of bandits, we have visualized the resulting topologies as network diagrams in Figure 6.5 (uniform selection), Figure 6.7 (an-

nealing epsilon) and Figure 6.6 (epsilon greedy $\epsilon = 0.2$). Each network diagram shows each replica as a vertex, colored by region e.g. purple is California, teal is Sao Paulo, Brazil, etc. Each vertex is also labeled with the 2-character UN country or US state abbreviation as well as the replica's precedence id. The size of the vertex represents the number of `Put` requests that replica received over the course of the experiment; larger vertices represent replicas that were colocated with workload generators. Each edge between vertices represents the total number of successful synchronizations, the darker and thicker the edge is, the more synchronizations occurred between the two replicas. Edges are directed, the source of the edge is the replica that initiated anti-entropy with the target of the edge.

Comparing the resulting networks, it is easy to see that more defined topologies result from the bandit-based approaches. The uniform selection network is simply a hairball of connections with a limited number of synchronizations. Clear optimal connections have emerged with the bandit strategies, dark lines represent extremely successful synchronization connections between replicas, while light lines represent synchronization pairs that are selected less frequently. We posit that fewer edges in the graph represents a more stable network; the fewer synchronization pairs that are selected, the less noise that occurs from selecting a peer that is in a similar state.

6.2 Bandits Discussion

To achieve stronger eventual consistency, the visibility latency of a system replicated with anti-entropy must be reduced. We believe that this can be achieved

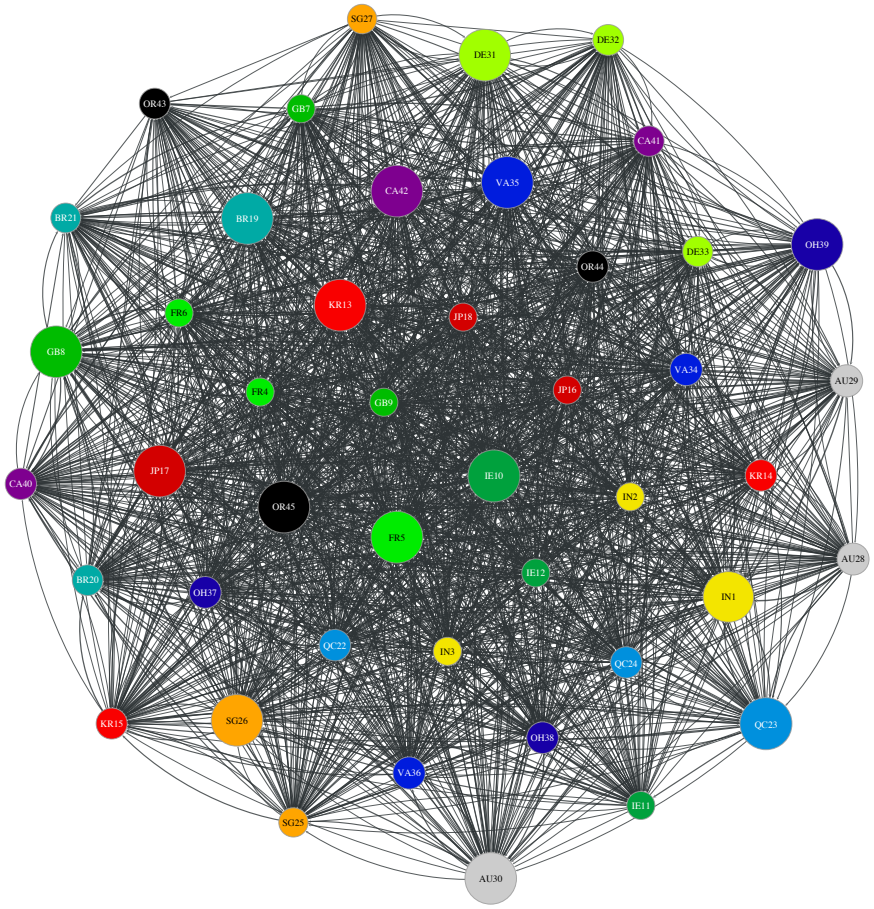

Figure 6.5: Synchronization network using uniform random selection of synchronization peers.

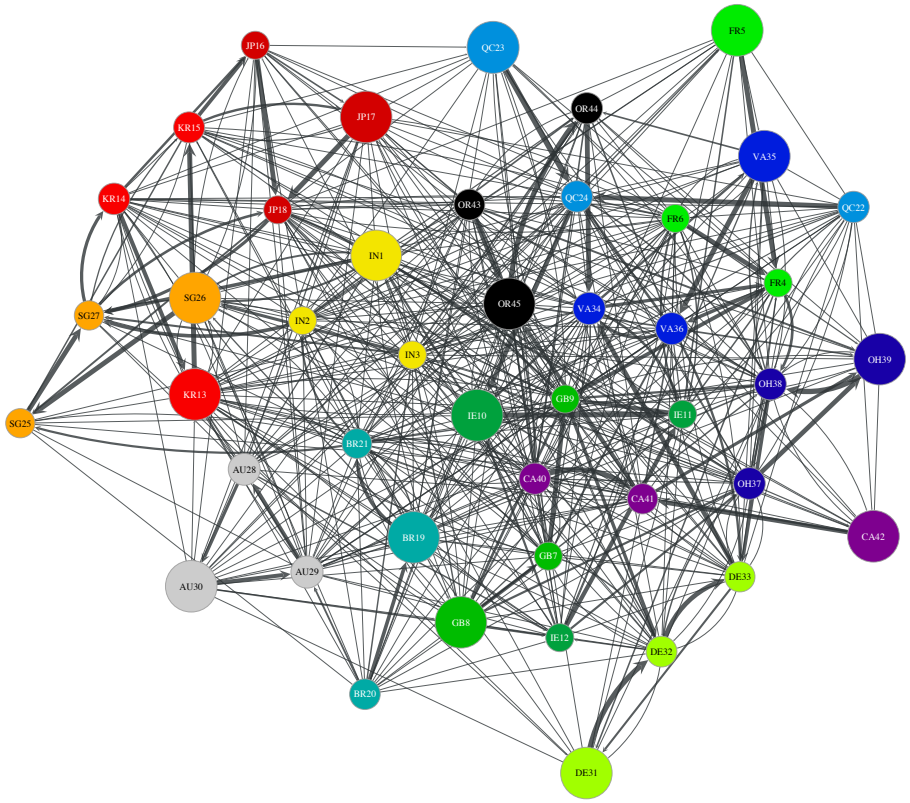

Figure 6.6: Synchronization network using bandit based selection of synchronization peers with $\epsilon = 0.2$.

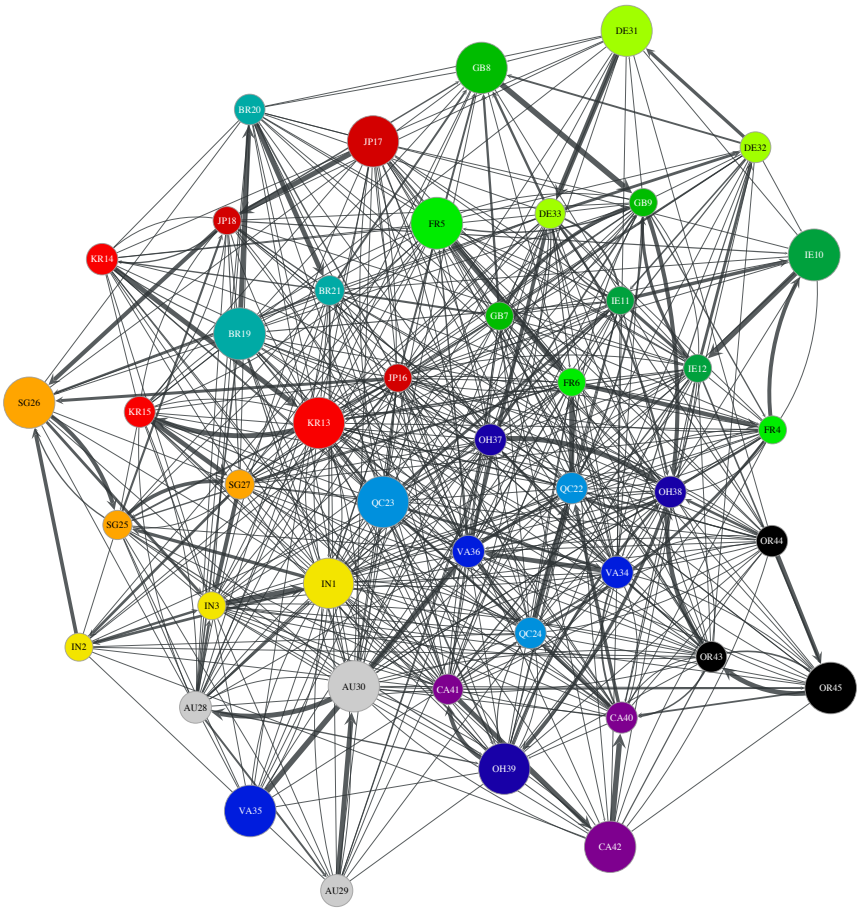

Figure 6.7: Synchronization network using annealing epsilon bandit based selection of synchronization peers.

with two primary goals: increasing the number of successful synchronizations and maximizing the number of local and regional synchronizations such that the average latency of anti-entropy sessions is as low as possible. These goals must also be tempered against other requirements, such as fault and partition tolerance, a deterministic anti-entropy solution that ensures the system will become consistent eventually, and load balancing the synchronization workload evenly across all replicas.

Bandit based approaches to peer selection clearly reduce noise inherent in uniform random selection as shown in Figure 6.3. The bandit strategies achieve better rewards over time because peers are selected that are more likely to have an update to synchronize. Moreover, based on the network diagrams shown in Figures 6.5-6.6, this is not the result of one or two replicas becoming primary syncs: most replicas have only one or two dark in-edges meaning that most replicas are only the most valuable peers for one or two other replicas.

Unfortunately, the rewards using a bandit approach, while clearly better than the uniform case, are not significantly better – this is an interesting demonstration of the possibility of adaptive systems to improve consistency but further investigation is required. The primary place we see for adjustment is future work to explore the reward function in detail. For example, the inclusion of penalties (negative rewards) might make the system faster to adjust to a high quality topology. Comparing reward functions against variable workloads may also reveal a continuum that can be tuned to the specific needs of the system.

As for localization, there does appear to be a natural inclination for replicas

that are geographically proximate to be a more likely selection. In Figure 6.6, replicas in Canada (light blue), Virginia (dark blue), Sydney (grey), California (purple), and Frankfurt (light green) all prioritize local connections. Regionally, this same figure shows strong links such as those between Ohio and California (`CA42` → `OH38`) or Japan and Singapore (`JP17` → `SG25`). Replicas such as `BR19` and `IN3` appear to be hubs that specialize in cross-region collaboration. Unfortunately there does also seem to be an isolating effect, for example Sydney (grey) appears to have no significant out of region synchronization partners. Isolated regions could probably be eliminated by scaling rewards with the number of transmitted updates, or by using larger epsilons. Multi-stage bandits might be used to create a tiered reward system to specifically adjust the selection of local, regional, and global peers. Other strategies such as upper confidence bounds, softmax, or Bayesian selection may also create more robust localization.

Finally, and perhaps most significantly, the experiments conducted in this paper were on a static workload; future work must explore dynamic workloads with changing access patterns to more closely simulate real world scenarios. While bandit algorithms are considered online algorithms that do respond to changing conditions, the epsilon greedy strategy can be slow to change since it prefers to exploit high-value arms. Contextual bandits use side information in addition to rewards to make selection decisions, and there is current research in exploring contextual bandits in dynamic worlds that may be applicable [92]. Other strategies such as periodic reseting of the values may incur a small cost to explore the best anti-entropy topology, but could respond to changing access patterns or conditions in a meaningful way.

Future efforts will consider different reward functions, different selection strategies, dynamic environments, and how the priorities of system designers can be embedded into rewards. Reward functions that capture more information about the expected workload of the system such as object size, number of conflicts, or localizing objects may allow specific tuning of the adaptive approach. We will also specifically explore in detail the effect of dynamic workloads on the system and how the reinforcement learning can adapt in real time to changing conditions. We plan to investigate periodic resets, anomaly detection, and auction mechanisms to produce efficient topologies that are not brittle as access patterns change. We also plan to evaluate other reinforcement learning strategies such as neural or Bayesian networks to determine if they handle dynamic environments more effectively.

6.3 Access Temperature Approaches

add this section

- Expected model of access patterns (daylight).

6.4 Other Types of Adaptation

add this section and conclusion

Monitor and Optimize.

Replica placement, object placement

6.5 Conclusion

In this chapter we have presented a demonstration of adaptive consistency in the geo-replicated eventually consistent systems by employing a novel approach to peer selection during anti-entropy – replacing uniform random selection with multi-armed bandits. Multi-armed bandits consider the historical reward obtained from synchronization with a peer, defined by the number of objects synchronized and the latency of RPCs, when making a selection. Bandits balance the exploitation of a known high-value synchronization peer with the exploration of possibly better peers or the impact of failures or partitions. The end result is a replication network that is less perturbed by noise due to randomness and capable of more efficiently propagating updates.

In an eventually consistent system, efficient propagation of updates is directly tied to higher consistency. By reducing visibility latency, the likelihood of a stale read decreases, which is the primary source of inconsistency in a highly available system. We have demonstrated that bandit approaches do in fact lower visibility latency in a large network.

We believe that the results presented show a promising start to a renewed investigation of highly available distributed storage systems in novel network environments, particularly those that span the globe. Specifically, this work is part of a larger exploration of adaptive, globally distributed data systems that federate consistency levels to provide stronger guarantees [93]. Federated consistency combines adaptive eventually consistent systems such as the one presented in this

paper with scaling geo-replicated consensus such as Hierarchical Consensus [42] in order to create robust data systems that are automatically tuned to provide the best availability and consistency. Distributed systems that adapt to and learn from their environments and access patterns, such as the emerging synchronization topologies we observed in this paper, may form the foundation for the extremely large, extremely efficient networks of the future.

Chapter 7: Related Work

Spanner [16] provides global consistency by sharding each tablet across multiple Paxos groups then externalizes their consistency using TrueTime, delaying the commit until a window of uncertainty has passed.

CalvinFS [18,19] batches transaction operations across the wide area, but still requires paxos to be deployed across the wide area.

Systems that implement many small quorums of coordination [16,17,94] avoid the centralization bottleneck and reliability concerns of master-service systems [71, 95] but create silos of independent operation that are not coordinated with respect to each other.

7.1 Hierarchical Consensus

Our principle contribution is Hierarchical Consensus, a general technique to compose consensus groups, maintain consistency invariants over large systems, and adapt to changing conditions and application loads. HC is related to the large body of work improving throughput in distributed consensus over the Paxos protocol [43, 48, 96, 97], and on Raft [49, 78]. These approaches focus on fast vs. slow path consensus, eliding phases with dependency resolution, and load balancing.

Our work is also orthogonal in that subquorums and the root quorums can be implemented with different underlying protocols, though the two levels must be integrated quite tightly. Further, HC abstracts reconfiguration away from subquorum consensus, allowing multiple subquorums to move into new configurations and reducing the need for joint consensus [49] and other heavyweight procedures. Finally, its hierarchical nature allows the system to multiplex multiple consensus instances on disjoint partitions of the object space while still maintaining global consistency guarantees.

The global consistency guarantees of HC are in direct contrast to other systems that scale by exploiting multiple consensus instances [12,16,17] on a per-object basis. These systems retain the advantage of small quorum sizes but cannot provide system-wide consistency invariants. Another set of systems uses quorum-based decision-making but relaxes consistency guarantees [11, 98, 99]; others provide no way to pivot the entire system to a new configuration [94]. Chain replication [100] and Vertical Paxos [51] are among approaches that control Paxos instances through other consensus decisions. However, HC differs in the deep integration of the two different levels. Whereas these approaches are top down, HC consensus decisions at the root level replace system configuration at the subquorum level, and vice versa.

Possibly the closest system to HC is Scatter [94], which uses an overlay to organize consistent groups into a ring. Neighbors can join, split, and talk amongst themselves. The bottom-up approach potentially allows scaling to many subquorums, but the lack of central control makes it hard to implement global re-maps beyond the reach of local neighbors. HC ties the root quorum and subquorums tightly together,

allowing root quorum decisions to completely reconfigure the running system on the fly either on demand or by detecting changes in network conditions.

We claim very strong consistency across a large distributed system, similar to Spanner [16]. Spanner provides linearizable transactions through use of special hardware and environments, which are used to tightly synchronize clocks in the distributed setting. Spanner therefore relies on a very specific, curated environment. HC targets a wider range of systems that require cost effective scaling in the data center to rich dynamic environments with heterogeneity on all levels.

Finally, shared logs have proven useful in a number of settings from fault tolerance to correctness guarantees. However, keeping such logs consistent in even a single consensus instance has proven difficult [71, 101, 102]. More recent systems are leveraging hardware support to provide fast access to shared logs [18, 19, 54, 56, 57, 103]. To our knowledge, HC is the first work to propose synchronizing shared logs across multiple discrete consensus instances in the wide area.

Chapter 7: Conclusion

Appendix A: Formal Specification

Will add formal specification here.

Bibliography

[1] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, and others. Oceanstore: An architecture for global-scale persistent storage. In *ACM Sigplan Notices*, volume 35, pages 190–201.

[2] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Elsevier.

[3] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium On*, pages 124–131. IEEE.

[4] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. 2014(239):2.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. A view of cloud computing. 53(4):50–58.

[6] Cisco Visual Networking Index. The zettabyte era–trends and analysis.

[7] Michael J. Casey and Paul Vigna. In blockchain we trust. 15:2018.

[8] Michael Stonebraker and Joey Hellerstein. What goes around comes around. 4.

[9] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: Friends or foes? 53(1):64–71.

[10] C. Mohan. History repeats itself: Sensible and NonsenSQL aspects of the NoSQL hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 11–16. ACM.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. 26(2):4.

[13] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. 44(2):35–40.

[14] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. pages 22–28.

[15] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, volume 11, pages 223–234.

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. Spanner: Google's globally distributed database. 31(3):8.

[17] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM.

[18] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM.

[19] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14.

[20] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. 33(2):51–59.

[21] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. 7(3):181–192.

[22] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. 45(2):37–42.

[23] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM.

[24] stevestein. Scaling out with Azure SQL Database.

[25] Alain Jobart, Sugu Sougoumarane, Michael Berlin, and Anthony Yeh. Vitess.

[26] Spencer Kimball, Peter Mattis, and Ben Darnell. CockroachDB.

[27] Erik Kain. The 'Pokémon GO' Launch Has Been A Complete Disaster [Updated].

[28] Cloud Datastore.

[29] Eric A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 167–167. ACM.

[30] Luke Stone. Bringing Pokémon GO to life on Google Cloud.

[31] Makiko Yamazaki. Developer of Nintendo's Pokemon GO aiming for rollout to 200...

[32] Richard George. Nintendo's President Discusses Region Locking.

[33] Dropbox — Company Info.

[34] Slack. Slack About Us.

[35] WeWork. Global Access.

[36] Tile About Tile.

[37] Stella Garber. Lessons Learned From Launching Internationally.

[38] Desdemona Bandini. RunKeeper Scales to Meet Demand from 24 Million Global Users with New Relic.

[39] Adam Grossman and Jay LaPorte. Dark Sky Weather App for iOS and Android.

[40] Matthew Hughes. Signal and Telegram are growing rapidly in countries with corruption problems.

[41] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154.

[42] Benjamin Bengfort and Pete Keleher. Brief Announcement: Hierarchical Consensus. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing*, pages 355–357. ACM.

[43] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM.

[44] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, volume 8, pages 369–384.

[45] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium On*, pages 111–120. IEEE.

[46] Pierre Sutra and Marc Shapiro. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium On*, pages 255–264. IEEE.

[47] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference On*, pages 156–167. IEEE.

[48] Leslie Lamport. The part-time parliament. 16(2):133–169.

[49] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319.

[50] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-stop Processors. 2(2):145–154.

[51] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, pages 312–313. ACM.

[52] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *OSDI*, volume 4, pages 8–8.

[53] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. 3(4):1.

[54] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 1–14. ACM.

[55] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D Davis. CORFU: A Shared Log Design for Flash Clusters. In *NSDI*, pages 1–14.

[56] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, and others. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *NSDI*, pages 35–49.

[57] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM.

[58] Leslie Lamport. Paxos made simple. 32(4):18–25.

[59] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. 4(3):382–401.

[60] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. 29.

[61] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference On*, pages 140–149. IEEE.

[62] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. 12(3):463–492.

[63] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. 12(2):91–122.

[64] David K. Gifford. Information Storage in a Decentralized Computer System.

[65] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 21(7):558–565.

[66] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. (3):240–247.

[67] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps-decentralized version vectors. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference On*, pages 544–551. IEEE.

[68] Philip A. Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 923–928. ACM.

[69] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. A brief history of the Internet. 39(5):22–31.

[70] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. 2:199–216.

[71] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM.

[72] Michael Stonebraker. The case for shared nothing. 9(1):4–9.

[73] Google. Google RPC.

[74] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM.

[75] Jon Gjengset <jon@thesquareplanet.com>. Students' Guide to Raft :: Jon Gjengset.

[76] Blake Mizerany, Qin Yicheng, and Li Xiang. Etcd: Package raft.

[77] Caio Oliveira, Lau Cheuk Lung, Hylson Netto, and Luciana Rech. Evaluating raft in docker on kubernetes. In *International Conference on Systems Science*, pages 123–130. Springer.

[78] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? 49(1):12–21.

[79] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

[80] Benjamin Bengfort, Philip Y. Kim, Kevin Harrison, and James A. Reggia. Evolutionary design of self-organizing particle systems for collective problem solving. In *Swarm Intelligence (SIS), 2014 IEEE Symposium On*, pages 1–8. IEEE.

[81] Benjamin Bengfort, Konstantinos Xirogiannopoulos, and Pete Keleher. Anti-Entropy Bandits for Geo-Replicated Consistency. In *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press.

[82] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. 5(8):776–787.

[83] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. 23(2):279–302.

[84] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 1. ACM.

[85] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: The Consumers' Perspective. In *CIDR*, volume 11, pages 134–143.

[86] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. 45(5):37–42.

[87] Salvatore Scellato, Cecilia Mascolo, Mirco Musolesi, and Jon Crowcroft. Track globally, deliver locally: Improving content delivery networks by tracking geographic social cascades. In *Proceedings of the 20th International Conference on World Wide Web*, pages 457–466. ACM.

[88] Bernhard Haeupler. Simple, fast and deterministic gossip and rumor spreading. 62(6):47.

[89] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium On*, pages 565–574. IEEE.

[90] Yamir Moreno, Maziar Nekovee, and Amalio F. Pacheco. Dynamics of rumor spreading in complex networks. 69(6):066130.

[91] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information. In *Advances in Neural Information Processing Systems*, pages 817–824.

[92] Haipeng Luo, Alekh Agarwal, and John Langford. Efficient Contextual Bandits in Non-stationary Worlds.

[93] Benjamin Bengfort and Pete Keleher. Federating Consistency for Partition-Prone Networks. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[94] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM.

[95] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM.

[96] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited.

[97] Leslie Lamport. Generalized consensus and Paxos.

[98] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. 1(2):1277–1288.

[99] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM.

[100] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, volume 4, pages 91–104.

[101] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association.

[102] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9.

[103] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR*, volume 11, pages 9–12.